Code to Career:

JavaScript & Node.js

from Zero to Professional

MERT TÜREDÜ

Code to Career: JavaScript & Node.js from Zero to Professional

First Edition, 2026

Errata & feedback: **mertturedu.thedev@gmail.com** (subject: `[Errata]`)

About the Cover

The squirrel on this book's cover wasn't chosen at random.

A squirrel getting ready for winter gathers thousands of seeds one by one — storing them, sometimes forgetting where, sometimes rediscovering them — but never stopping. What makes it resilient isn't one big leap; it's a small habit repeated day after day: one more seed today, one more tomorrow.

Becoming a developer works much the same way. You don't become "senior" overnight; every small concept you learn, every small bug you fix, every line of code you read adds up over time. **Curiosity** pulls you toward a new topic, **diligence** helps you go deeper into it, **patience** keeps you from giving up where you don't understand, and **preparation** means trusting that what you learn today will matter tomorrow — much like the squirrel storing seeds without knowing exactly when winter will arrive.

By picking up this book, you're already starting to gather your own seeds. Some chapters won't click on the first pass — the squirrel doesn't find every seed on the first try either. What matters is that you keep gathering.

Work well, and be patient — when winter comes, you'll be ready.

— Mert Türedü

Foreword

My first JavaScript code was an error message.

Where I meant to type `console.log`, I had typed `console.lo`. The terminal spat out something red, and I closed it without knowing what it meant. After that day, it took me another five years to learn how to read error messages, because nobody had taught me to read them — they had only told me what to do.

This book is here to close that gap.

Anyone who wants to learn JavaScript today has hundreds of resources in front of them: YouTube videos, blog posts, interactive platforms, official documentation. They are all well-intentioned. But none of them are connected to one another. From one you learn `var`, in another you read why `let` was invented, and in a third you're told to use `const`. There's no single voice tying the differences together.

I tried to build that connection. A resource that tells the whole road — from starting at zero to writing production-ready code — in a single voice, in a single order. It begins by asking "What does a computer actually do?" and goes on to explain why async/await was invented, what TypeScript really solves, and why Node.js's event loop looks so strange. It also tells the history of each topic, because knowing why something exists sticks in your mind far longer than learning how to use it.

Who is this for?

If you've never written code: Start from PART ZERO and don't skip anything. Those chapters may look "too easy," but if the foundation doesn't settle, everything after it starts to wobble.

If you know some JavaScript: Skim the first chapters quickly. Chapter 4 (functions), Chapters 11–13 (async), and Chapter 14 (OOP) are where you'll really want to stop.

If you're coming to JS from another language: Be prepared for the absence of a type system and for `this` behaving very differently than it does in Java. Chapter 4's `this` section (4.7) and the TypeScript chapter were written for you.

If you're refreshing your knowledge: The Cheat Sheets at the end of each chapter and Chapter 29 are enough.

If you're preparing for interviews: Go straight to §A.1–§A.10 (Algorithmic Thinking). These chapters teach interview-focused algorithmic thinking — patterns like Big O, Two Pointers, Binary Search, and DP — from scratch. Take the diagnostic test before entering the chapter; if you can pass that test, you can skip this chapter.

How should you read it?

In order. This book is linear; each chapter builds on the previous one. Skipping ahead is tempting, I know. But when you skip Chapter 3, you'll find yourself saying "I still don't get it" in Chapter 8, and what you'll have to go back to is Chapter 3.

Do the exercises. Writing teaches, not reading. When you see an exercise, try it yourself first; if you're stuck for at least 10 minutes, look at the solution.

Don't rush. Chapters 11–13 in particular (asynchronous programming) are the chapters most readers have to read three times. That's normal. I read them three times too.

Run every code example. Seeing the output teaches you five times more than reading does.

Look up any term you don't recognize. If you come across terms like `runtime`, `scope`, or `callback`, check **Chapter 28 — Glossary** at the end of the book. New terms are explained where they first appear; still, if something doesn't look familiar, the Glossary is always there.

The Structure of the Book

Every chapter draws from a common pool: a one-sentence summary of the topic, its history (why was it invented?), a short memorable formula, an in-depth explanation, a diagram, exercises, and an end-of-chapter cheat sheet. Not all of these appear in every chapter; if the topic doesn't call for it, or it's already covered elsewhere, that piece is simply absent. You can do the same thing as you read: if you already know the history, skip it; if a look at the cheat sheet is enough, move on.

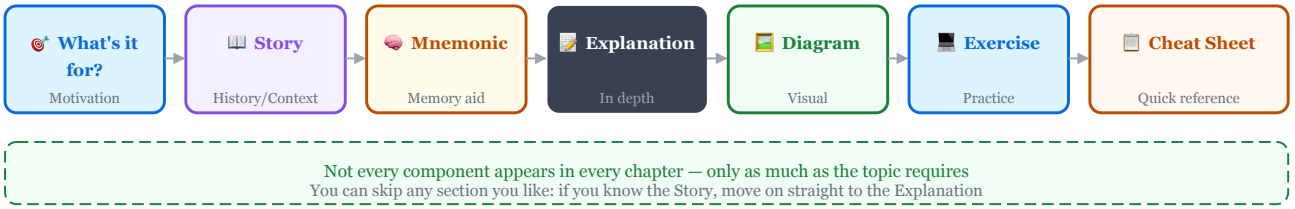


FIGURE 0.0 *The Pedagogical Structure of Each Chapter*

Not all of these components appear in every chapter — only as much as the topic requires. Think of them as a **menu**: read what's useful to you, skip what you find unnecessary. If you already know a topic, you can skip the Story and go straight to the Explanation or the Cheat Sheet. The familiar structure is meant to make the book *easy to navigate*, not to force you into a single way of reading.

Choose your reading mode

It wouldn't be right to have everyone read the book at the same pace. Depending on where you stand, I recommend three modes:

MODE	WHO IT'S FOR	HOW
🦕 **Full reader**	Never written code; wants to build a solid foundation	Read every chapter in order, don't skip the story or the mnemonics, do every exercise
🚶 **Normal reader**	Knows a little programming	Skim the story and mnemonics quickly, move on to the explanation, do half of the exercises
🚴 **Fast scan**	Knows another language, exploring JS	Read the chapter intro, its tables, and the cheat sheet; only the exercises that catch your interest

Chapters labeled `ADVANCED` **and** `EXPERT` cover specialized topics; you can skip them on a first read and come back later. For the full explanation of each chapter's label, see the **How Are Chapters Marked?** table below.

You can skip the story, mnemonic, and cheat sheet sections if they aren't useful to you for that topic. You don't have to read the history of something you already know — jump straight to the code example.

A Note

Learning to write code can sometimes be frustrating. You get errors, something doesn't work, you get stuck for hours. This is a **natural** part of the process—not a sign that you're a bad developer, but a sign that you're really learning. Experienced developers go through the same thing; they just know what to do when they get stuck. This book aims to give you both JavaScript and that "knowing what to do" skill.

Happy reading.

Errata & Feedback

In a book of this size, a few errors are inevitable, incorrect code output, a typo, a misleading explanation. I would be grateful if you reported every error you find to **mertturedu.thedev@gmail.com**. Writing `[Errata]` in the subject line keeps your message from getting lost; if you add which chapter, which page, and a short explanation, the correction gets made much faster.

Feedback goes to the same address: you can write in about a topic you'd like explained better, a heading you feel is missing, or an example you had trouble understanding.

🗺️ Learning Journey

This book has a linear structure; each chapter builds on the previous one. The table below summarizes the entire book at a glance; for each row you can see **what you'll learn** and **a small example** you can try right away in your terminal.

CHAPTER	TOPIC	WHAT YOU'LL LEARN	TRY IT NOW
Z.1	Programming logic	What an algorithm is; how computer instructions are processed	`node -e "console.log(1 + 1)"`
Z.2	Editor · Terminal · Browser	The three core development tools and the differences between them	`ls` → `node file.js` → F12
Z.3	Environment setup	Installing Node.js and VS Code; running your first file	`node --version`
Z.4	Your first program	Variables, operations, output; a working program in four lines	`const x = 5; console.log(x * 2)`
Z.5	Mental Model	Variable = labeled box; function = input→machine→output	`let n = 42; console.log(typeof n)`
Z.6	Your First Conditional and Loop	Making decisions with if/else; repeating with for; conditional looping with while	`if (age >= 18) { ... } else { ... }`
Ch. 0	Introduction	How you'll read the book; reading modes; chapter structure	*(read, not run)*
Ch. 0.5	Debug Mindset	Reading error messages; SyntaxError / ReferenceError / TypeError; the REPL loop	Write intentionally buggy code and read the message
Ch. 1	JavaScript History & V8	How JS was born; what V8 is; JIT compilation; Node.js = V8 + libuv	`node -e "console.log(process.version)"`
Ch. 2	Variables & Types	`const`, `let`, `var`; 7 primitive types; `typeof`; `===`; falsy; scope; hoisting; NaN; coercion	`typeof null` → `"object"` (historical bug)
Ch. 3	Operators	`?.`, `??`, `||`, `&&`, ternary; the pitfalls of modern operators	`user?.address?.city ?? "unknown"`
Ch. 4	Functions	Parameters, return, arrow fn, closure, first-class	`const multiply = (a, b) => a * b`
Ch. 4.2	Functional Programming	Pure function, immutability, map/filter/reduce, currying	`[1,2,3].filter(x => x > 1).map(x => x * 2)`
Ch. 5	Object	`{ }` syntax, access, spread, destructuring; the basics of the prototype chain	`const { name } = user`
Ch. 6	Array	`map`, `filter`, `reduce`, spread, destructuring	`[1,2,3].map(x => x * 2)`
Ch. 7	Map & Set	Dictionary (Map) and unique set (Set)	`new Set([1,1,2])` → `{1, 2}`
Ch. 8	String & Regex	`split`, `join`, `trim`, `includes`, regex basics	`"  ali  ".trim()` → `"ali"`
Ch. 9	Flow Control	`if/else`, `switch`, ternary, early return	`x > 0 ? "positive" : "negative"`
Ch. 10	Error Handling	`try/catch/finally`, error types, `throw`	`throw new Error("Invalid input")`
A.1	Algorithms & Big O	Time/space complexity; intuition for O(1), O(n), O(log n)	`[1,3,5,7,9].indexOf(7)` → O(n)

CHAPTER	TOPIC	WHAT YOU'LL LEARN	TRY IT NOW
A.2	Two Pointers	From both ends toward the middle in a sorted array: reduce $O(n^2)$ to $O(n)$	palindrome, Two Sum sorted
A.3	Sliding Window	A window that slides over a contiguous subarray/string: O(n)	longest substring without repeats
A.4	Hash Map Patterns	O(1) lookup; "have I seen this before?"; frequency counter	Two Sum with `new Map()`
A.5	Stack & Queue	LIFO/FIFO; bracket matching; queue for BFS	is `()[]{}` valid?
A.6	Linked List	Pointer chain; fast & slow; reversal	cycle detection
A.7	Binary Search	Halve the sorted space: O(log n)	search in a rotated array
A.8	Tree: DFS & BFS	Recursive DFS; BFS with a queue; BST validation	level-by-level traversal
A.9	Graph & Union-Find	Adjacency list; island count; merging components	Number of Islands
A.10	Dynamic Programming	Memoization; tabulation; subproblem cache	Fibonacci → Coin Change
Ch. 11	Promise	Asynchronous chaining, `.then`, `.catch`, `.finally`	`fetch(url).then(r => r.json())`
Ch. 12	async / await	Writing asynchronous code as if it were synchronous	`const data = await fetch(url)`
Ch. 13	Event Loop (attention)	Why `setTimeout(fn, 0)` waits; the microtask queue	`console.log` ordering test
Ch. 14	OOP & Class	`class`, `constructor`, `extends`, `super`; the prototype chain underneath	`class Animal { sound() {} }`
Ch. 15	CommonJS	`require`, `module.exports`, `__dirname`; node_modules resolution	`const fs = require("fs")`
Ch. 16	ES Modules	`import / export` (ESM); default vs named export; tree-shaking	`export const PI = 3.14`
Ch. 17	Node.js Core	`fs`, `path`, `process`, Buffer, stream basics	`fs.readFileSync("a.txt", "utf8")`
Ch. 18	HTTP & REST API	`http.createServer`, Express, routes, middleware	`app.get("/", (req, res) => ...)`
Ch. 19	Crypto & HMAC	Hash, HMAC, JWT token generation; encryption basics	`crypto.createHash("sha256")...`
Ch. 20	AsyncLocalStorage	Request context tracking; trace ID; carrying context automatically	Add a trace ID to every log line
Ch. 21	Streams & SSE	Large data streaming, server-sent events, backpressure	`res.write("data: hello\n\n")`
Ch. 22	Design Patterns	Singleton, Factory, Observer, Strategy	Recognizing them in real Node.js projects

CHAPTER	TOPIC	WHAT YOU'LL LEARN	TRY IT NOW
Ch. 23	Firestore & NoSQL	Document-based DB; collection/query; SQL vs NoSQL difference	`db.collection("users").add({...})`
Ch. 24	Cloud & Docker	Deployment, containers, environment variables	`docker build -t app .`
Ch. 24.1–24.2	Security & Performance	SQL injection, XSS, rate limiting, profiling	`helmet()` middleware
Ch. 24.3–24.14	Observability & DevOps	Logging, tracing, CI/CD, Kubernetes	Structured log format
Ch. 25	Project Structure	Folder organization, code cleanliness, DRY	Monolith → modular refactor
Ch. 26.1	TypeScript	Static types, `interface`, generics, utility types	`const x: number = 5`
Ch. 26.2–26.4	Testing	Unit (Vitest), integration, the TDD loop	`expect(add(1,2)).toBe(3)`
§B0–B11	Browser & DOM	DOM manipulation, Fetch API, Web Worker	`document.querySelector("#btn")`
§F1–F15	Frontend Ecosystem	React, Vue, Next.js, bundler (Vite)	`<button onClick={...}>Click</button>`

> 📎 **This table shows the book's main backbone.** The **§A.1–§A.10 Algorithmic Thinking** sections sit between §10 and §11 — LeetCode-style problem solving, Big O, data structures, and classic algorithms. Other important sections not listed in the table: **Chapter 18.3** (WebSocket), **Chapter 18.4** (GraphQL & gRPC), **Chapter 23.11** (PostgreSQL & SQL), **Chapter 23.1** (Prisma / Drizzle ORM), **Chapter F9** (Bun & Deno). See the **Contents** for the full list.

🛠️ **Mini Projects** (throughout the book): CLI Calculator → Async Data Fetcher → HTTP Echo Server → JWT Auth → Realtime Chat → Final: Production-ready full-stack application.

> **Async + Event Loop** (Chapters 11–13) is where most readers get stuck. Don't rush; read it twice if you need to.

🏷️ How Are Chapters Labeled?

Below each chapter heading, you'll find a **difficulty label** and **estimated reading time**:

LABEL	MEANING	ESTIMATED TIME
BEGINNER	Requires no prior knowledge; everyone should read it	15–30 min
BASIC	If you've read previous chapters, you're ready	20–40 min
INTERMEDIATE	Requires focus and some practice; don't rush	30–50 min
ADVANCED	Complex topic; can be skipped on first pass; see the ROADMAP box	30–60 min
EXPERT	Optional deep dive; only if you specifically need it	30–60 min

First reading strategy. Read the labeled chapters in order. When you reach a difficult chapter, don't panic: chapters like these start with a 🧭 **ROADMAP** box. That box tells you three things: (1) what is the prerequisite for this topic, (2) whether you should read it now or save it for later, (3) where you'll pick up if you skip it. Postponing a topic isn't failure; it's learning in the right order.

Chapters with "X" (Chapter 2.1, 4X, 6X, 11X…) are **deep-dive appendices** to the main narrative, separating advanced details of a topic from the main flow. You can skip them on first read; return when you actually start using that topic. The "X" in the heading means "this is a deep dive, not required."

🌳 Skill Tree

The map below shows how the skills you'll learn connect to one another. Each box builds on top of the previous level, so progress without skipping ahead.

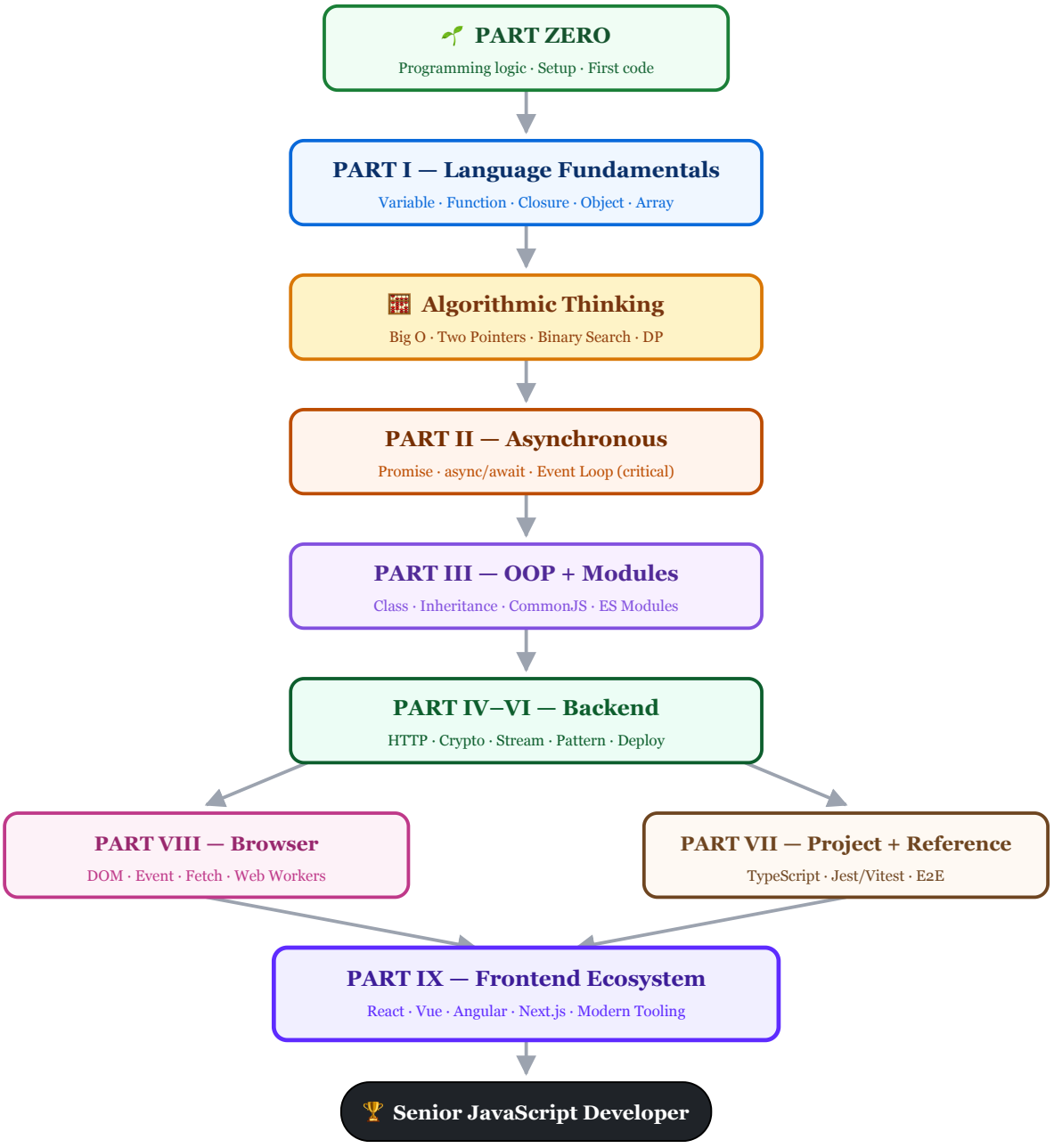


FIGURE 0.1B *JavaScript & Node.js Skill Tree: From Zero to Senior*

Contents

CODE TO CAREER · JAVASCRIPT & NODE.JS FROM ZERO TO PROFESSIONAL

152 CHAPTERS

PART ZERO – FIRST STEPS

Chapter Z.1 – How Does a Computer Think?

● ○ ○ ○ ○ BEGINNER

> ◎ WHAT IS IT GOOD FOR?
> Understanding the fundamental logic of programming: a computer reads a recipe and carries it out. Writing code is nothing more than writing that recipe.

Most people working with a computer for the first time think of it as a "smart assistant." The truth is the exact opposite — and far more interesting: a computer is the world's fastest but most *literal* apprentice. It does exactly what you say; both what you leave out and what you add. In this chapter you'll learn how to describe a task to that apprentice.

The computer does exactly what you tell it

FIGURE Z.1.1 *The computer: fast but obedient; you must state every instruction explicitly*

Computers are extremely fast — they perform billions of operations per second. But on their own they can make no decisions. Two fundamental rules apply:

1. **They do everything you tell them.**
2. **They do nothing you don't tell them.**

Why are these rules so strict? Because computers know no context; they cannot infer "what you meant." You tell a human assistant "make coffee," and they infer from context what you had in mind. A computer cannot interpret context; every step must be written out explicitly.

This is the essence of programming: expressing your intent **step by step** and **precisely**. In natural language, "make coffee" is enough; to a computer you have to say:

OUTPUT

```
1. Put water in the pot
2. Light the stove
3. Wait for the water to boil
4. Add the coffee to the water
5. Stir for 30 seconds
6. Pour into the cup
```

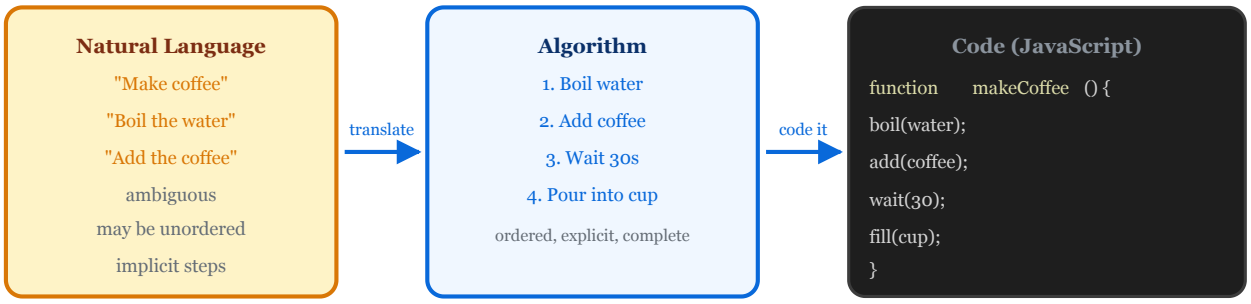


FIGURE Z.1.2 *An algorithm: everyday recipe → numbered steps → code*

This list is an **algorithm**. Writing code means translating it into a language the computer can understand.
We wrote the recipe; but in what *order* does the computer process these steps?

Commands run top to bottom

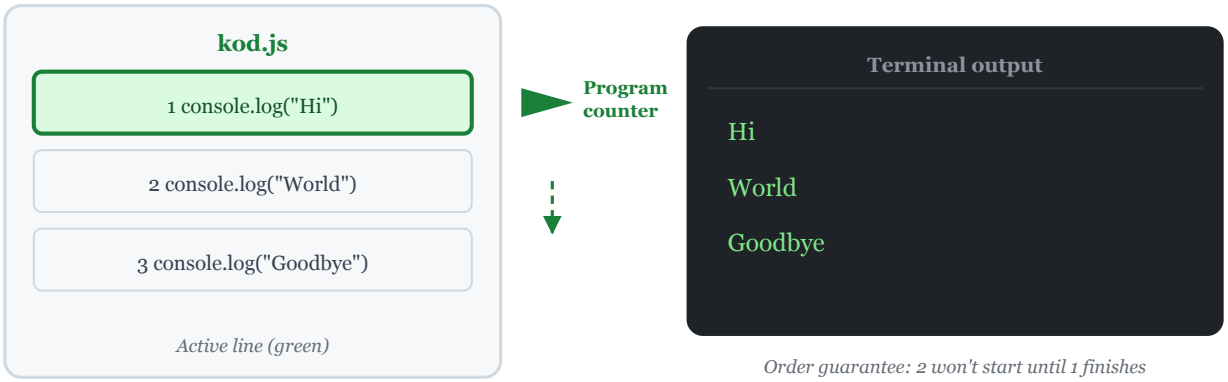


FIGURE Z.1.3 *Sequential execution: the program counter (pointer) processes each line in order*

In JavaScript, each line is interpreted in order:

```js
console.log("Hi");
console.log("World");
console.log("Goodbye");
```

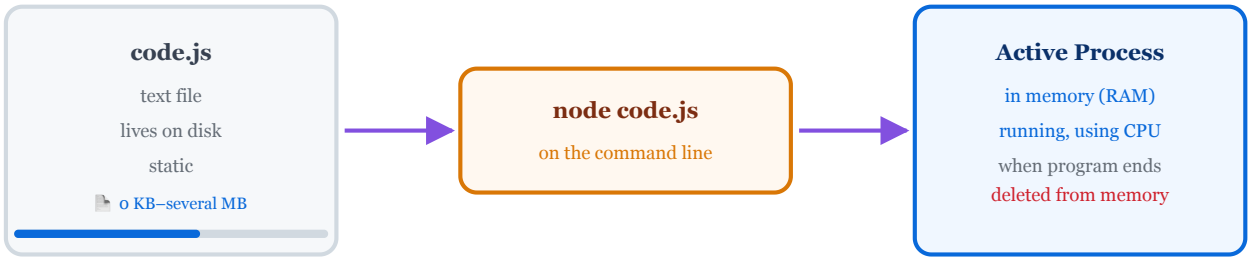
When you run it:

```
Hi
World
Goodbye
```

The first line finishes, then execution moves to the second line. This behavior is called **sequential execution**, and it is the most fundamental model of programming. Later, we'll compare it with **concurrent execution** (Chapters 11–13).

There's one more subtlety: the file you write and the thing that runs are not the same.

Code is not the same as a running program



Recipe (code) stays — cake (program) is eaten, nothing left on the table

FIGURE Z.1.4 *Code (disk) → node command → Program (RAM): the difference between a recipe and a baked cake*

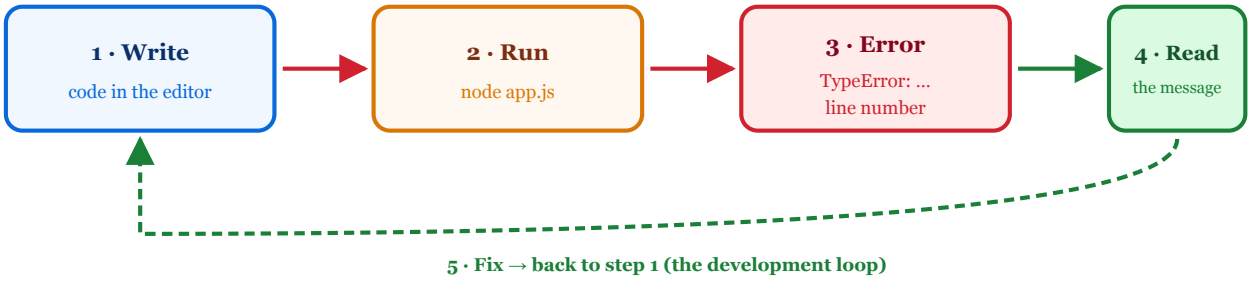
This difference is easy to miss while learning:

- **Code**: a text file, static. Lives on disk.
- **Program**: the moment that code is running. Lives in memory, disappears when it ends.

```
[ code.js ]  ——— run with node ———→  [ active process ]
   text                                  memory + CPU
   on disk                               temporary
```
SCHEMA

Like the difference between a recipe card and a baked cake. The card stays, the cake is eaten, nothing left on the table.

Bug = the mistake in the recipe



Getting bugs is normal — this loop repeats throughout your career

FIGURE Z.1.5 *The bug loop: get an error → read it → fix it → run again*

The computer does **exactly** what you write. If something isn't working the way you expect, that's not bad news — it's good news. Because the mistake wasn't made by the computer, it was made by you, and you have the power to fix it.

```js
console.lo("Hi");
```
JS

JavaScript error:

```
TypeError: console.lo is not a function
```
ERROR

The computer is telling you this: "The console object has no function called `lo`. Did you mean `log`?" You fix it, you move on. This loop — writing and testing, getting an error, fixing it — is the daily routine of development.

The word "bug" has an interesting origin: in 1947, Grace Hopper and her team, working on the Harvard Mark II computer, came across an actual moth inside the machine while hunting down the source of an error. In their logbook they wrote "first actual case of bug being found"; that record is on display today at the Smithsonian National Museum of American History. Ever since, software errors have been called "bugs," and hunting them down "debugging."

So, what language do we use to explain these recipes to the computer?

What is a programming language?

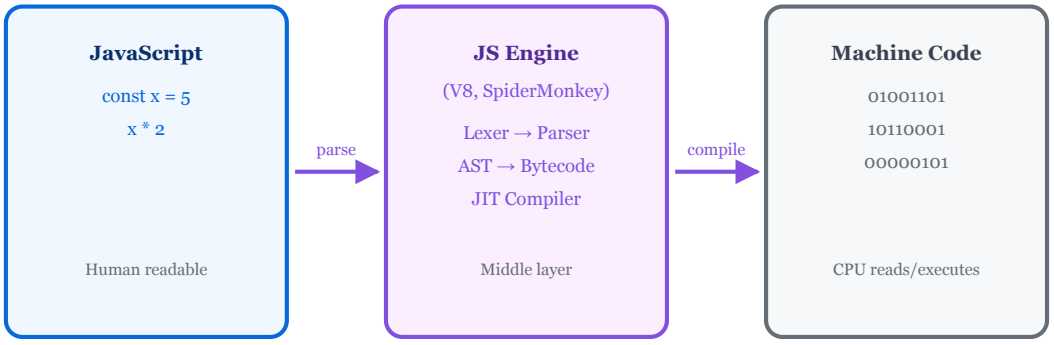


FIGURE Z.1.6 *Abstraction layers: human code → JS engine → machine code → CPU*

The language a computer actually speaks is **machine code** (0s and 1s). Writing programs in this language is nearly impossible for a human. That's why intermediate languages were developed:

LANGUAGE	COMMON USE CASES
JavaScript	Web browsers, servers, mobile, desktop
Python	Data science, machine learning, automation
Go	System services, high concurrency
Rust	Performance-critical systems, a safe C alternative
Swift	Apple platforms (iOS, macOS)
Kotlin	Android, JVM-based applications

Every language has its own syntax and philosophy, but the core logic is the same: converting human-readable code into machine-executable form.

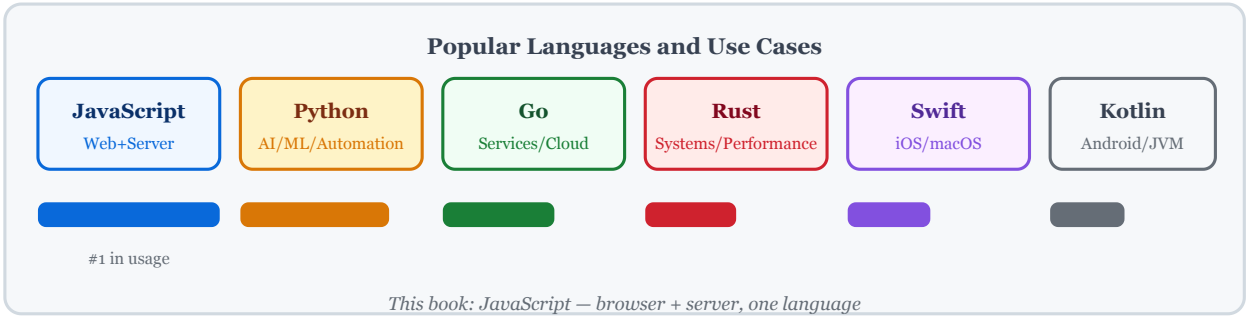


FIGURE Z.1.7 *Use cases of popular languages*

This book's subject is **JavaScript**. It's the most widely used language in the world; at first it ran only in browsers, and starting in 2009 (with Node.js) it began running on servers as well. Today, being able to write both frontend and backend in a single language is an advantage in its own right.

– STORY – ALAN TURING AND COMPUTABILITY

*But is there a problem a computer cannot solve? In 1936, the mathematician Alan Turing asked exactly this question — and answered it. The theoretical model he developed, the **Turing machine**, formed the conceptual foundation of all modern computers. Turing also mathematically proved the question "which problems can be solved with an algorithm, and which cannot?" This work defines the very reason programming languages exist: transferring human thought — at least a certain category of it — to a machine. When you write JavaScript, you are standing on top of this theoretical foundation.*

Error Culture

Beginners are typically afraid of errors. For professional developers, it's different: seeing an error means gaining knowledge. Learning to read errors is more important than learning syntax.

In later chapters, we'll cover error handling (Chapter 10) and the "debugging mindset" (Chapter 0.5) separately. For now, this intuition is sufficient:

> An error is feedback your programming language gives you. Silence—silent failure—is more dangerous.

What this chapter covers

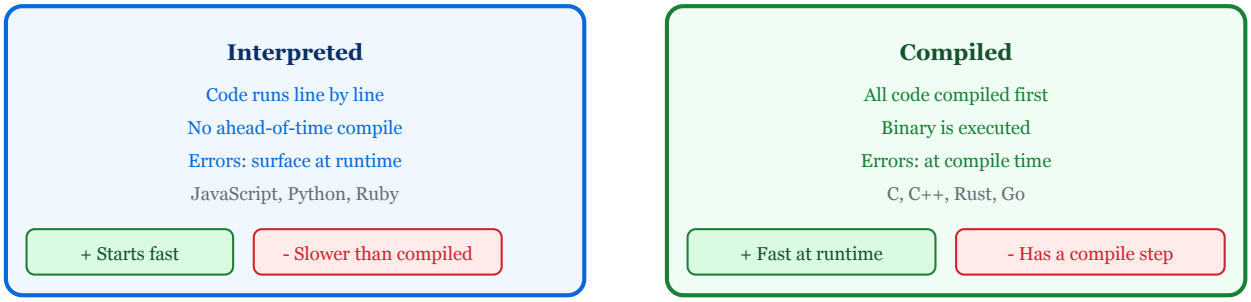


FIGURE Z.1.8 *Interpretation vs Compilation: JS is interpreted (sped up with JIT)*

CONCEPT	SUMMARY
Algorithm	A step-by-step method for solving a problem
Sequential execution	Instructions run in order, top to bottom
Code vs Program	Code = text, Program = running instance
Bug	An error in the code you write — it's normal and unavoidable
Programming language	The interface between human and computer

CHEAT SHEET CHAPTER Z.1

CONCEPT	DEFINITION
Computer	Fast but obedient and blindly compliant; it needs every instruction spelled out explicitly
Program	A sequence of instructions given to the computer
Bug	An error in the recipe you wrote; not the computer's fault
Algorithm	A step-by-step method for solving a problem

REVIEW QUESTIONS: CHAPTER Z.1

1. Translate the coffee recipe below from plain language into "programming logic" (as numbered steps): "*First boil the water. Once the water boils, put the coffee in the cup. Pour the boiling water into the cup. Stir.*"
2. Why doesn't the computer ask you the "what should I do?" question?
3. What does the code below print?

```js
console.log("a");
console.log("b");
console.log("a");
```

4. Bug = ?
5. Recipe vs. cake — what's the equivalent of code vs. program in JavaScript?

Now you know that the computer waits for instructions; in the next chapter we move on to the three tools you'll use to write those instructions: the editor, the terminal, and the browser.

▼ VIEW ANSWERS

1. Example: `1. Boil the water. 2. Put the coffee in the cup. 3. Pour the water over it. 4. Wait 3 minutes. 5. Fill the cup.` — Each step is clear, ordered, and does a single job.
2. The computer applies the given instructions exactly; it doesn't figure out which step is missing or wrong. It isn't someone who asks you what to do — it's a tool that does whatever you tell it.
3. `hello` — `console.log` prints the value inside the parentheses to the screen.
4. Bug: an error that breaks the intended behavior of the program. The process of finding and fixing errors is called debugging.
5. Recipe = code (the instructions), the baked cake = program output (the running result). When JavaScript code runs, a program comes into being.

Chapter Z.1: How Does a Computer Think? — Quiz Questions

1. Which is the fundamental operating rule of a computer?

A) It can make decisions on its own
B) It does exactly what it's told, and nothing it isn't told
C) It automatically corrects faulty instructions
D) It can understand natural language
E) It picks the most logical step

2. How is an algorithm most accurately defined?

A) A step-by-step written sequence of instructions for solving a specific problem
B) The processor inside the computer
C) The name of a programming language
D) The software that makes code run
E) A list where faulty code is collected

3. Which property must an algorithm absolutely have?

A) It must consist of at least 10 steps
B) It only applies to math problems
C) Every step must contain precise instructions with no ambiguity
D) The steps must be kept as short as possible
E) All steps must be able to run at the same time

4. A student runs the code `sonuç = 10 / 0`. What does JavaScript do in this case?

A) It rejects the operation as a mathematical rule and halts
B) It automatically returns 0
C) It returns `Infinity`; it does not throw an error
D) It asks the user about the operation
E) It only shows a warning and produces no result

5. What is the difference between "code" and a "program"?

A) They are the same thing, just different names
B) Code is static text (a recipe), a program is the running process (the baked cake)
C) Code is machine language, a program is written in human language
D) A program is shorter than code
E) Code is only used for JavaScript

6. What happens if we give a computer the instruction `make coffee`?

A) It searches the internet for a coffee recipe
B) It guesses the most logical steps
C) It asks for the steps
D) It runs a coffee machine connected to the system
E) It errors out or runs incorrectly because it doesn't recognize the command

7. What is Alan Turing's fundamental contribution to programming?

A) He invented the first programming language
B) He built the first computer
C) He mathematically defined which problems can be solved by a computer
D) He coined the term bug
E) He laid the foundation of JavaScript

8. Which of the following steps is in the wrong order in the `boil water` algorithm?

`1. Add water → 2. Light the stove → 3. Wait for the water to boil → 4. Pour into the cup → 5. Put coffee in the pot`

A) Steps 1 and 2 should swap
B) Step 2 should be last
C) Step 3 should be skipped
D) Steps 4 and 5 should swap; the coffee goes in first
E) The order is correct

9. Which statement best sums up the essence of programming?

A) Thinking about what you want the computer to do
B) Making the code look pretty
C) Expressing your intent step by step and precisely
D) Understanding error messages
E) Memorizing the programming language

10. When a programmer writes `total = 5 + 3`, what does the computer do?

A) It compares 5 and 3
B) It asks the user to fix the error
C) It prints 5 + 3 to the screen
D) It writes the value 8 into the memory cell named `total`
E) It performs the operation in the future

▼ ANSWERS AND EXPLANATIONS

1 → B
`A computer does exactly what it's told, and nothing it isn't told` is the book's most fundamental rule. A computer has neither intuition nor empathy; it only executes clear, ordered instructions. The other options anthropomorphize the computer.

2 → A
An algorithm is a set of instructions written in a specific order to solve a problem, each step being clear and finite. It is independent of any programming language and can even be written on paper.

3 → C
The most fundamental property of an algorithm is that every step is **precise and free of ambiguity**. Not "boil the water" but "heat the water until it reaches 100°C." An ambiguous instruction → an ambiguous result. Step count, subject area, or concurrency are not mandatory properties of an algorithm.

4 → C
In JavaScript, the operation `10 / 0` returns `Infinity`; it does not throw an error. The computer doesn't say "this is meaningless"; it does exactly what it's told. Languages like Python throw a `ZeroDivisionError`. This difference shows that each language defines its own division-by-zero behavior separately. `console.log(10 / 0)` → `Infinity`, `console.log(-10 / 0)` → `-Infinity`, `console.log(0 / 0)` → `NaN`.

5 → B
Code is written instructions, like a recipe. A program is the process that comes to life when that code is executed, like the baked cake. When the same code is run multiple times, each run creates a separate program instance (process).

6 → E
A computer doesn't understand natural language. `make coffee` is an ambiguous instruction; you need to write out each step separately and clearly. An ambiguous instruction → either an error or a wrong result.

7 → C

With his theory of computability, Alan Turing mathematically established which problems can be solved algorithmically and which cannot. This is considered the theoretical foundation of computer science.

8 → D

In step 4, `pour into the cup` is done, but the coffee hasn't been added yet. The correct order is: `add the coffee → stir → pour into the cup`. In an algorithm, order is critical; the wrong order produces the wrong result.

9 → C

Programming is the discipline of turning intent into step-by-step, precise instructions that a computer can understand. Pretty code or debugging is a result of this, not its essence.

10 → D

The computer performs the operation `5 + 3` to produce 8 and assigns this value to the memory cell (variable) named `total`. To print it to the screen, you would additionally need `console.log(total)`.

PART ZERO – FIRST STEPS

Chapter Z.2 – Editor, Terminal, Browser: Three Friends

● ○ ○ ○ ○ BEGINNER

> ◎ WHAT IS IT GOOD FOR?
>
> Getting to know the three fundamental tools a developer uses all day long: the editor where you **write** code, the terminal where you **run** code, and the browser where you **see** the result.

Think of this trio as a team: one *writes* (editor), one *runs* (terminal), one *displays* (browser). Three friends you'll bounce back and forth between all day long.

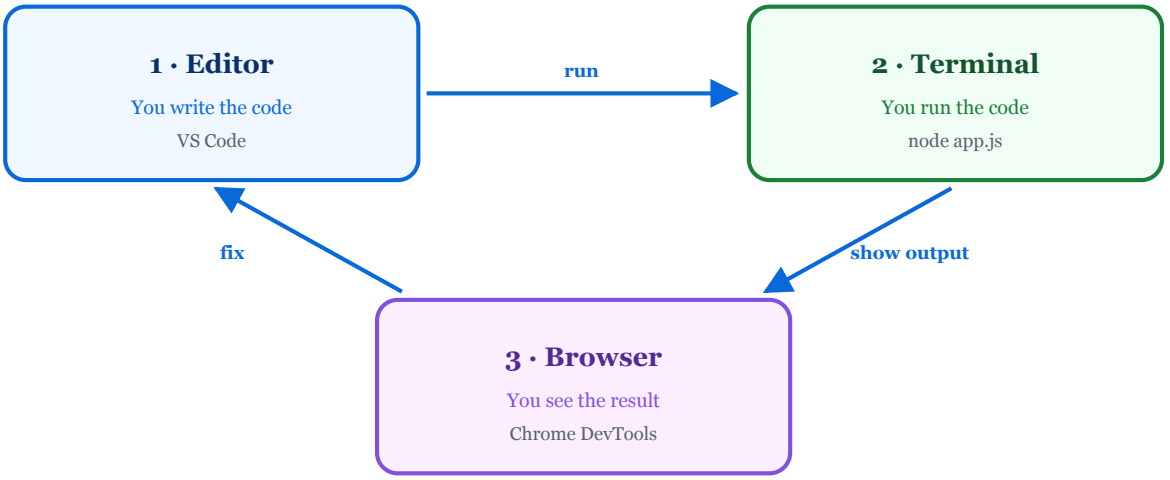


FIGURE Z.2.1 *The development loop: Write → Run → See*

1. Editor: Where you write your code

FIGURE Z.2.2 *The VS Code interface: sidebar, editor, terminal, and syntax coloring*

An editor (a code-writing application) is different from a word processor. Writing code in Microsoft Word isn't practical, for these reasons:

- There's no syntax highlighting (everything is black).
- It doesn't offer smart completion (autocomplete).
- The spell checker wrongly flags your code.
- It inserts invisible formatting characters.

Instead, you use a **code editor**. The most common choice, **Visual Studio Code** (VS Code for short), is free and open source, developed by Microsoft.

What VS Code gives you:

FEATURE	BENEFIT
Syntax highlighting	`function` is blue, strings are green; code becomes more readable
Smart completion	Type `arr.` and a list like `map, filter, ...` pops up
Error underline	Shows invalid syntax with a red wavy line
Integrated terminal	You can open a shell inside the editor
Git integration	Displays file changes visually
Extension ecosystem	Thousands of extensions like Prettier, ESLint, GitLens

Other popular editors: WebStorm (JetBrains, paid but powerful), Sublime Text (lightweight), Neovim (terminal-based, for advanced users).

The friend who writes your code is set — but a text file won't move on its own. Now the friend who *runs* it steps onto the stage.

2. Terminal: Where Commands Are Run

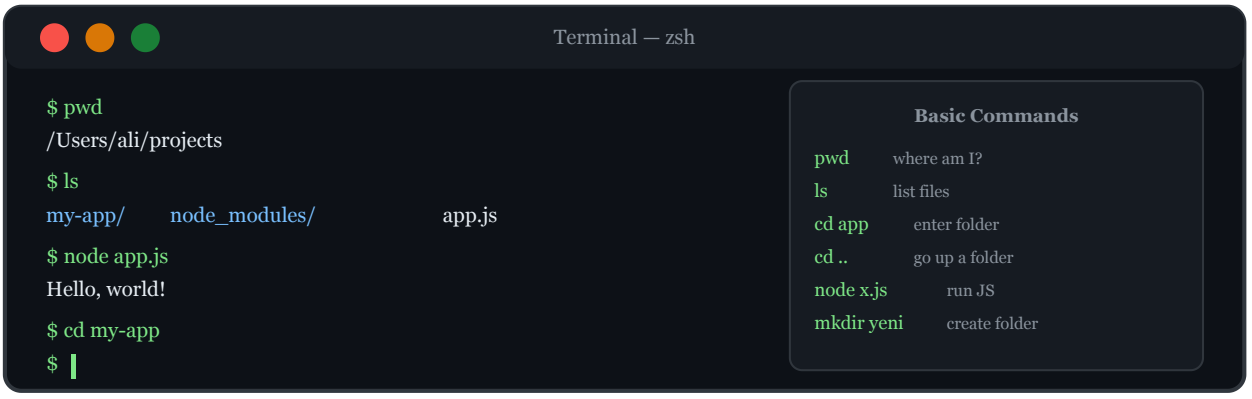


FIGURE Z.2.3 *Basic terminal commands: ls, cd, pwd, node*

You can think of the terminal as a Finder/Explorer you navigate with the keyboard: everything you'd click on there, you do here by typing.

The terminal (or "command line") is the **window** where you interact with the computer using written commands. The shell, on the other hand, is the **interpreter** running inside that window: the program that understands and executes your commands. On macOS/Linux the most common shells are `bash` or `zsh`; when you open a terminal, a shell is running in the background. Before graphical interfaces (GUIs), computers could only be used through the terminal. It's still used today because:

- Automation is easy (you can script it).
- It's the only option for connecting to remote servers (SSH).
- It can do things a GUI can't (special git commands, network analysis, etc.).

Basic commands:

```
ls              # List the files in your current directory
cd folder       # Switch to the folder directory
cd ..           # Go up one directory
pwd             # Which directory am I in right now?
mkdir new       # Create a new directory
rm file.txt     # Delete a file
mv old.js new.js   # Rename or move
cp a.txt b.txt  # Copy a file
cat file.txt    # Show the contents of a file
```

TERMINAL

The most critical command for JavaScript:

```
node script.js
```

TERMINAL

This line runs the JavaScript code in your `script.js` file. The output (for example, `console.log(...)`) is written to the terminal.

Where do you open the terminal?

- **macOS**: `Cmd+Space` → type "Terminal" and hit Enter
- **Windows**: Start → "PowerShell" or "Windows Terminal"
- **Linux**: `Ctrl+Alt+T` on most distributions
- **Inside VS Code**: `Ctrl+`` (backtick) opens the integrated terminal

The one that writes and the one that runs are ready; now it's the turn of the friend that *displays* the result.

3. The Browser: Where the Result Is Displayed

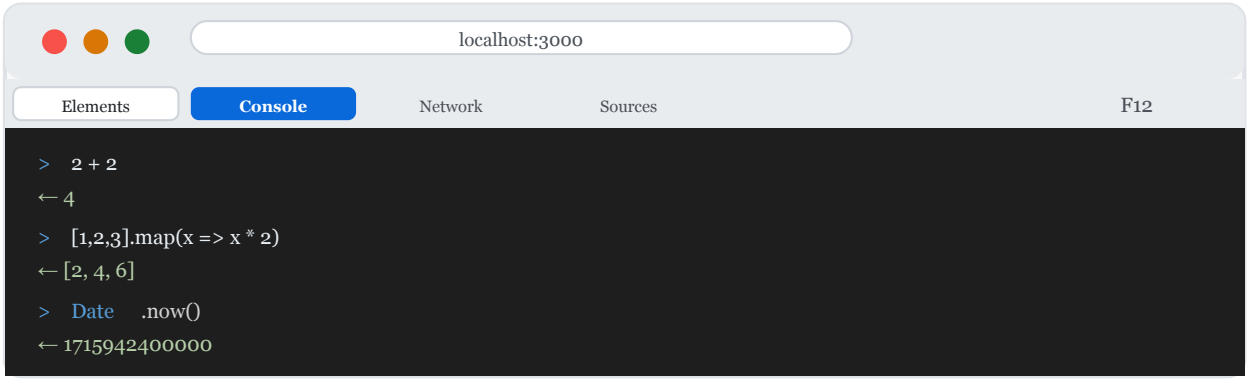


FIGURE Z.2.4 *Browser DevTools Console: opened with F12, you can run JS instantly*

The browser (Chrome, Firefox, Safari, Edge) doesn't just display web pages — it's a **JavaScript runtime environment**. Behind every web page you open, JavaScript code is running.

Tim Berners-Lee wrote the first browser in 1990 at CERN; today the phone in your pocket carries a runtime environment far beyond that first dream.

The browser has two uses:

1. Viewing your web application: If you're building a website, the result appears in the browser.

2. Testing JavaScript (DevTools): Pressing `F12` (Windows/Linux) or `Cmd+Option+I` (Mac) opens the **Developer Tools**. In the Console tab you can run JavaScript instantly:

```
> 2 + 2
4
> "welcome".length
7
> Date.now()
1715942400000
```

JS

This is extremely handy for quick experiments. You can think of the browser console as a "calculator for JavaScript."

Development cycle: 3 tools together

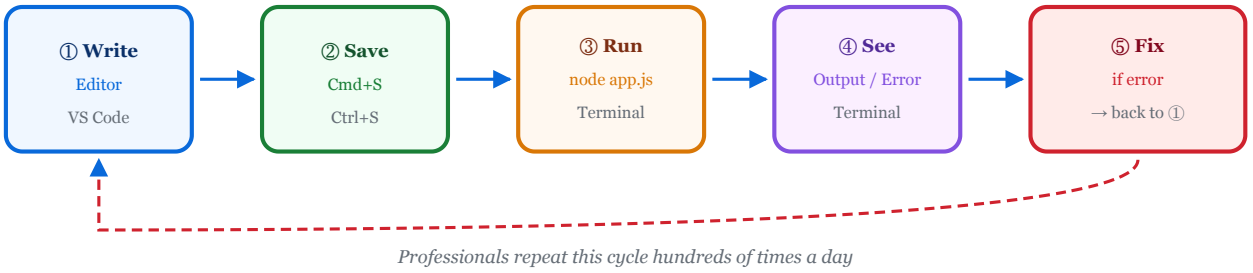


FIGURE Z.2.5 *The full development cycle: write → save → run → see → fix*

A typical development session consists of these steps:

STEP	TOOL	ACTION
1	Editor	Write or edit the code
2	Editor	Save the file (`Cmd+S` / `Ctrl+S`)
3	Terminal	Run it with `node app.js`
4	Terminal	See the output or the error
5	(if needed) Browser	View the web result
6	Editor	Fix it, go back to step 2

Professional developers repeat this cycle hundreds of times a day. That's why keeping all three windows open at once becomes a habit.

Try it: In VS Code, create a new file named `merhaba.js`. Inside it, write `console.log("My first program is running!")`, save (`Cmd+S` / `Ctrl+S`), open the terminal, and run the command `node merhaba.js`. See the output. You've completed the cycle.

When working with three friends, they all need a common language: the address of the files.

File path (path) concept

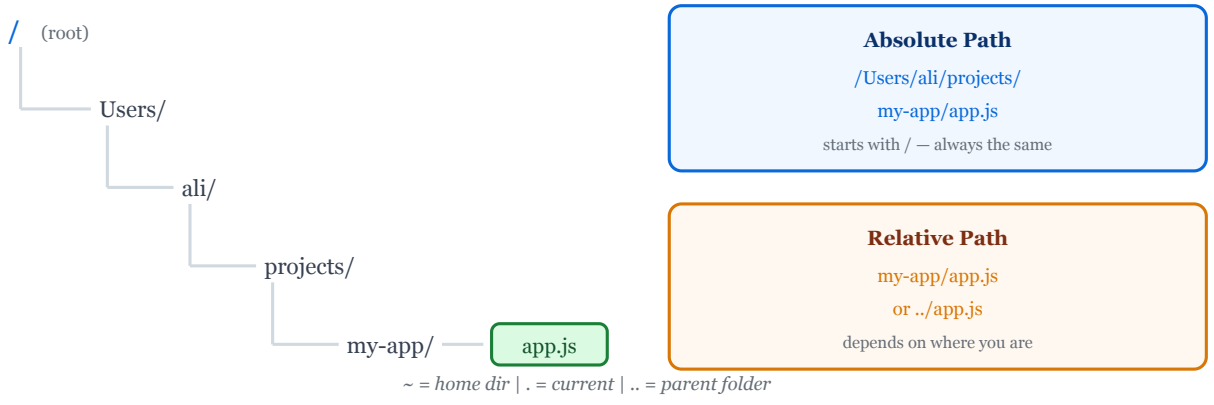


FIGURE Z.2.6 *File system tree: absolute path vs relative path*

Every file on your computer has a **path**. The path specifies the file's location within the folder structure:

DIRECTOR

```
/Users/ali/projects/my-app/src/app.js
└──┘ └──┘ └──────┘ └────┘ └─┘ └──┘
  │    │      │      │    │    │
 root home projects app  src  file
```

There are two kinds of paths:

- **Absolute path**: `/Users/ali/projects/my-app/src/app.js` : starts from the root.
- **Relative path**: `src/app.js` : relative to the directory you're currently in.

Terminal commands mostly use relative paths:

› TERMINAL

```
cd ~/projects/my-app    # ~ is a shortcut for the user directory
node src/app.js         # relative to the directory I'm in
```

SYMBOL	MEANING
`~`	User home directory (`/Users/ali` or `/home/ali`)
`.`	The directory I'm in
`..`	One directory up
`/`	Root directory

CHEAT SHEET

CHAPTER Z.2

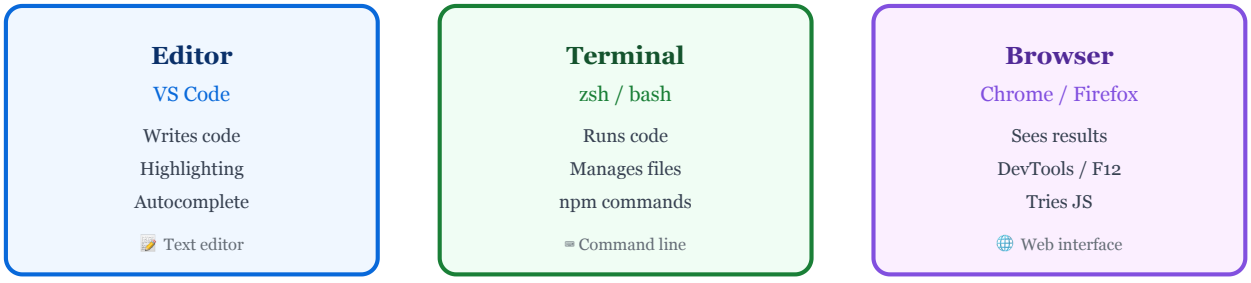


FIGURE Z.2.7 *Editor vs Terminal vs Browser: the role of each tool*

TOOL	ITS JOB
Editor (VS Code)	Where you **write** the code
Terminal	Where you **run** commands (`node app.js`)
Browser	Where you **see** the result (for the web)
`cd`	Change folder
`ls` (Mac) / `dir` (Win)	List folder contents
`pwd`	Where am I right now

REVIEW QUESTIONS: CHAPTER Z.2

1. What is the editor's main job? Does it run code?
2. What happens when you type `node server.js` in the terminal?
3. What does the `cd /Users/zeynep/Desktop` command do?
4. How do you open the browser DevTools (shortcut on Mac)?
5. Write the development loop in your own words.

Now that you know the tools, the next step is to install them on your own computer and run your first program.

▼ VIEW ANSWERS

1. The editor's main job is to write code; it edits text and highlights the syntax. It does not run the code — the one that runs the code is Node.js or the browser.
2. The `node server.js` command starts the Node.js interpreter, which reads and runs the `server.js` file. If there is an error in the file it prints an error message; otherwise it carries out whatever the code does.
3. The `cd /Users/zeynep/Desktop` command changes the current working directory in the terminal — it switches to Zoe's desktop folder.
4. On Mac: `Cmd + Option + I` or `Cmd + Option + J` (for the Console). Alternative: right-click → "Inspect" → DevTools opens.
5. The development loop: Write code → Save → Run (`node file.js`) → See the output → Fix the error → Run again. This loop repeats for every feature.

Chapter Z.2 — Editor, Terminal, Browser: The Three Friends — Quiz Questions

1. What is the fundamental job of a code editor (VS Code)?

A) To run code
B) To write and edit code
C) To display web pages
D) To host the JavaScript engine
E) To upload files to the server

2. What happens when you type `node server.js` in the terminal?

A) server.js opens in the browser
B) The file opens for editing in VS Code
C) The file is uploaded to the server
D) The Node.js runtime executes the server.js file
E) The file's contents are printed in the terminal

3. What is the shortcut to open Chrome DevTools on a Mac?

A) Cmd+Option+I
B) Cmd+D
C) F5
D) Ctrl+Shift+I
E) Ctrl+Alt+C

4. Which order is the correct development cycle?

A) Run → Write → See → Fix
B) See → Write → Run → Fix
C) Fix → Write → Run → See
D) Write → See → Run → Save
E) Write → Run → See → Fix if there's an error

5. When backend (server) Node.js code is executed, where is the output seen?

A) In the terminal
B) On the browser screen
C) In VS Code's editor area
D) In the Chrome DevTools Console
E) It's written to the end of the file

6. What shortcut is used to open the integrated terminal in VS Code?

A) Ctrl+T
B) Cmd+Enter
C) Ctrl+` (backtick)
D) Alt+T
E) Cmd+Shift+P

7. Who developed the first web browser, and at which institution?

A) Bill Gates, Microsoft
B) Brendan Eich, Netscape
C) Ryan Dahl, Google
D) Linus Torvalds, University of Helsinki
E) Tim Berners-Lee, CERN

8. What does the command `cd /Users/zeynep/Desktop` do in the terminal?

A) Deletes the Desktop folder
B) Changes the terminal's active directory to the Desktop folder
C) Copies the Desktop folder
D) Lists the files on the Desktop
E) Creates the Desktop folder

9. In the Editor, Terminal, and Browser trio, what is each one's role, respectively?

A) Run, Write, See
B) Save, Compile, Publish
C) See, Write, Run
D) Write, Run, See
E) Edit, Test, Publish

10. Where does frontend JavaScript code run?

A) In the terminal
B) Inside VS Code
C) In the browser's JavaScript engine
D) In Node.js
E) In the file system

▼ ANSWERS AND EXPLANATIONS

1 → B
An editor is solely a tool for writing and editing code. It does **not** run code. Running is the terminal's job (Node.js), and displaying is the browser's job.

2 → D
The Node.js runtime reads the server.js file and executes the JavaScript code line by line. The output appears in the terminal. The browser is not involved in this process.

3 → A
On a Mac, `Cmd+Option+I` opens Chrome DevTools. On Windows/Linux it's `F12` or `Ctrl+Shift+I`. DevTools gives you the ability to inspect the JavaScript running in the browser, debug it, and view network requests.

4 → E
The correct cycle: **Write (editor) → Run (terminal) → See (terminal/browser) → Fix if there's an error → Run again**. This cycle repeats throughout your career.

5 → A
A Node.js program is run in the terminal (with the `node file.js` command), and `console.log` output appears in the terminal. The browser only runs frontend code delivered over HTTP.

6 → C
`Ctrl+`` (backtick) opens VS Code's integrated terminal. This shortcut is the same on Mac and Windows. The integrated terminal lets you work directly in the editor without opening a separate terminal application.

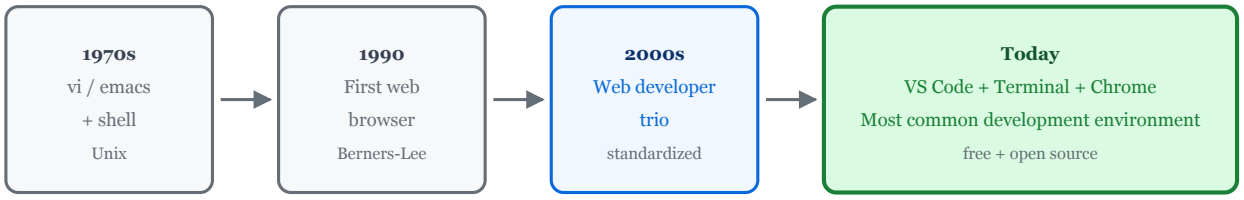
7 → E
Tim Berners-Lee wrote the first graphical web browser at CERN in 1990 (named WorldWideWeb). He also developed the HTTP protocol and the HTML language. He is regarded as the inventor of the web.

8 → B
The `cd` (change directory) command changes the active working directory in the terminal. After that, commands like `node file.js` run in this directory.

9 → D
Editor (VS Code) → you **write** the code. Terminal → you **run** the code. Browser → you **see** the result. This trio is the foundation of the web development cycle.

10 → C
Frontend JavaScript runs in the JavaScript engine in the user's browser (V8 in Chrome, SpiderMonkey in Firefox). Backend JavaScript, on the other hand, runs in the Node.js environment on the server.

Historical Background: The Evolution of Tools



We'll use this trio throughout this book

FIGURE Z.2.0 *History of developer tools: vi/emacs → shell → browser → VS Code + Chrome*

PART ZERO – FIRST STEPS

Chapter Z.3 – Setting Up Your Development Environment

●○○○○ BEGINNER

> ◎ WHAT IS IT GOOD FOR?
>
> Installing the two essential pieces of software your computer needs to run JavaScript: Node.js (the runtime) and VS Code (the editor). By the end of this chapter you'll be able to get a "node --version" output in the terminal and run your first piece of code.

– THE STORY

*In 2009, Ryan Dahl found a way to take the V8 JavaScript engine that Google had developed for Chrome, separate it from the browser, and run it directly on the operating system. The result: **Node.js**. Until that day, JavaScript lived only in the browser; thanks to Node.js, it became runnable everywhere — on servers, in command-line tools, in desktop applications. Today, Node.js is one of the most widely used runtimes in the world.*

Let's be honest: setup is the least fun part of programming — and it's also where beginners give up the most. The good news: it's a one-time investment of about 20 minutes; after that, it's code for a lifetime. If something goes wrong, don't panic — the troubleshooting table below is there for exactly that.

Step 1: Installing Node.js

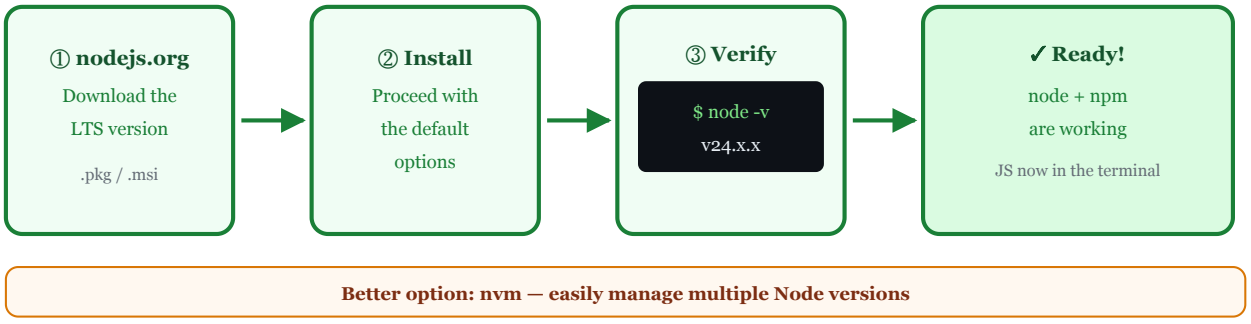


FIGURE Z.3.1 *Node.js installation flow: download → install → verify*

Go to nodejs.org ↗ and download the **LTS** (Long-Term Support) version. The LTS version is the stable release suitable for use in production. As this book was written, the current LTS is Node 24.

Installation:

- **macOS / Windows**: Open the `.pkg` or `.msi` file you downloaded and proceed with the default options.
- **Linux**: Use your distribution's specific package manager (`apt` , `dnf` , `pacman`), or manage versions with nvm ↗.

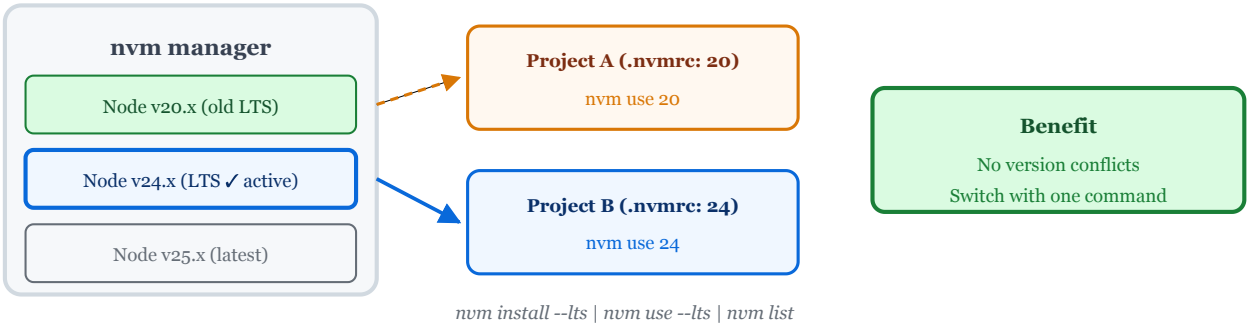


FIGURE Z.3.2 *Version management with nvm: project A v20, project B v24*

Better alternative: `nvm` (Node Version Manager): Lets you easily manage multiple Node versions. Down the road, different projects may want different Node versions.

```
# Installing Node with nvm
curl -o- https://raz.githubusercontent.com/nvm-sh/nvm/v0.39.7/install.sh | bash
nvm install --lts
nvm use --lts
```

Verification:

```
node --version    # v24.x.x
npm --version     # 10.x.x
```

If the `node` and `npm` (Node Package Manager) commands work, the installation was successful.

Node is installed — the engine is ready. Now let's set up the workbench where you'll write your code.

Step 2: VS Code installation

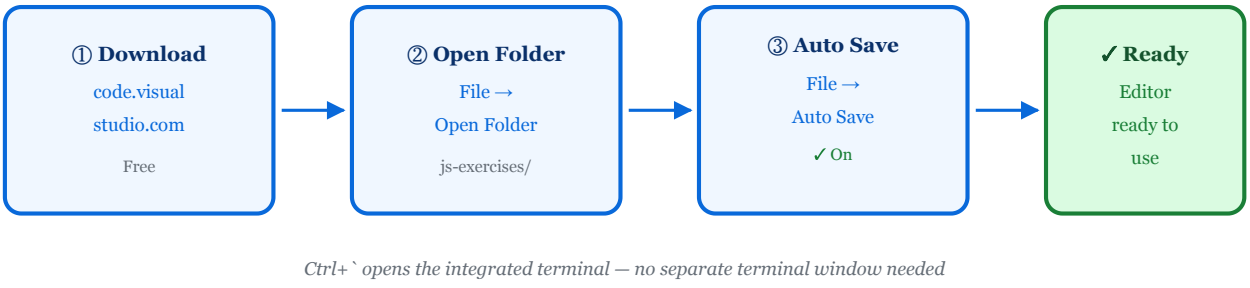


FIGURE Z.3.3 *VS Code first-launch steps: open folder → auto save → language setting*

Download the version for your operating system from code.visualstudio.com ↗ and install it.

Things to do on first launch:

1. **Open a folder**: choose an empty folder with `File → Open Folder` (e.g. `Desktop/js-exercises`).
2. **Turn on auto save**: `File → Auto Save`. Changes are saved automatically.

Step 3: Your First JavaScript File

In VS Code:

1. Press `Cmd+N` (macOS) or `Ctrl+N` (Windows) to open a new file.
2. Type this code:

```js
console.log("Hello, world!");

const name = "Alex";
const age = 25;
console.log(`${name}, ${age} years old.`);
```

3. Save it: `Cmd+S` / `Ctrl+S`, filename `first.js`.

Important: The file extension **must be** `.js`. VS Code recognizes the file type from the extension and applies syntax highlighting accordingly.

Step 4: Running the code

Ctrl+` backtick → VS Code integrated terminal

FIGURE Z.3.4 *Running the first program: file → terminal → output*

Open VS Code's integrated terminal: `Ctrl+`` (backtick, in the top-left corner of the keyboard, the key to the left of the 1).

Type this into the terminal:

```
node first.js
```

› TERMINAL

Output:

```
Hello, world!
Alex, age 25.
```

OUTPUT

🎉 Your first JavaScript program ran. That counts as a small win — you successfully completed the entire development cycle (write → save → run → see the output).

Did something go wrong? It goes wrong for everyone — no panic. Here are the spots people get stuck on most often and how to fix them:

Common installation problems

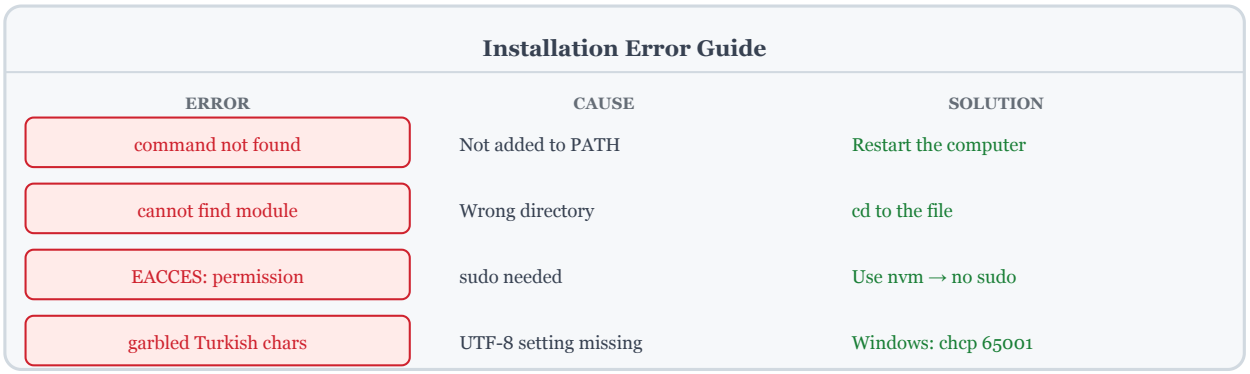


FIGURE Z.3.5 *Common installation errors and solutions*

ERROR	CAUSE	SOLUTION
`node: command not found`	Not added to PATH or installation incomplete	Restart your computer; if that doesn't fix it, reinstall Node
`cannot find module './first.js'`	You're in the wrong directory	Use the `cd` command to switch to the folder where the file is
Turkish characters garbled in the terminal	UTF-8 setting missing	No issue on macOS/Linux; on Windows, try `chcp 65001`
Turkish characters garbled in VS Code	Wrong file encoding	The bottom-right corner should say "UTF-8"; if not, click it and change it
`EACCES: permission denied` (npm install)	Sudo needed (bad approach)	Instead of this warning, use nvm so you don't need sudo

Recommended VS Code Extensions

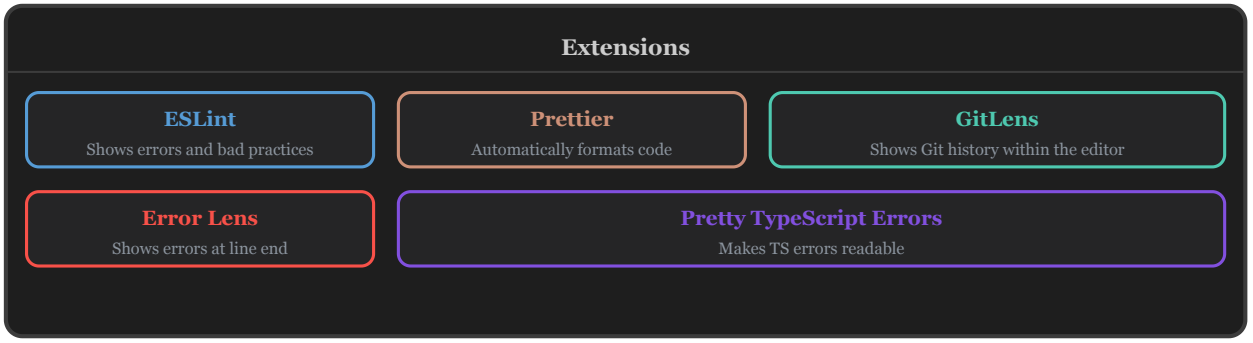


FIGURE Z.3.6 *Essential VS Code extensions and their functions*

Click the **Extensions** icon on the right edge of VS Code (four squares) and search for and install these:

EXTENSION	FUNCTION
ESLint	Shows possible errors and bad practices in code with red underline
Prettier	Automatically formats code (consistent indentation, line length)
GitLens	Lets you view Git history within the editor
Error Lens	Shows error messages directly at line end
Pretty TypeScript Errors	Makes TS errors readable

Bonus: pnpm and bun

> ▪ NOTE
> For a detailed comparison with **Bun** and **Deno**, the Node.js alternatives, see Chapters F9 and F13. For now, standard `npm` + Node.js is enough.

Later on you'll get to know **pnpm** (npm's disk-friendly alternative) or **bun** (a very fast new *runtime* — that is, the environment that runs your code, an alternative to Node.js). For now, standard `npm` is enough, but keep these in mind as a note:

› TERMINAL

```
# pnpm
npm install -g pnpm

# bun
curl -fsSL https://bun.sh/install | bash
```

We cover modern tooling in detail in Chapter F13.

CHEAT SHEET CHAPTER Z.3

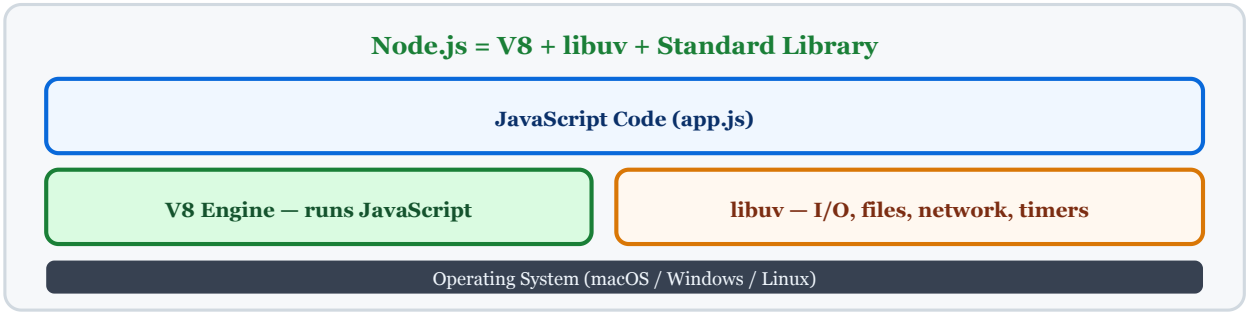


FIGURE Z.3.7 *Node.js architecture: V8 engine + libuv + standard library*

STEP	COMMAND
Node version	`node --version`
First file	`echo "console.log('hi')" > app.js`
Run	`node app.js`
Install package	`npm install <name>`
Init project	`npm init -y`

REVIEW QUESTIONS: CHAPTER Z.3

1. What does `node --version` do?
2. How do you open a terminal in VS Code?
3. How do you run a `.js` file?
4. Are `console.log("Hello")` and `console.log('Hello')` different? (Answer: no, in JS `'` and `"` are the same.)
5. Did you run your first program? (Ask yourself this and keep going.)

The environment is ready; in the next chapter you'll write a real program and try out variables and console commands with your own hands.

▼ VIEW ANSWERS

1. `node --version` shows the version number of the installed Node.js (e.g. `v24.1.0`). If it isn't installed, it returns an error.
2. In VS Code: `Ctrl + ``` (backtick) or from the top menu, Terminal → New Terminal. On Mac, `Cmd + ``.
3. With the `node fileName.js` command. In the terminal, you first `cd` into the folder where the file is located, then run this command.
4. No, they're not different. In JavaScript, single quotes `'` and double quotes `"` can be used interchangeably to define a string. What matters is that the opening and closing quotes match.
5. A personal-experience question — after you run your first `console.log("Hello World")`, you can say yes.

Chapter Z.3: Setting Up Your Development Environment – Quiz Questions

1. What does the `LTS` option mean when downloading Node.js?

A) Latest TypeScript Support; the version with TypeScript support
B) Lite Terminal Setup; the lightweight installation version
C) Long-Term Support; the stable version that receives long-term support
D) Linux Terminal Standard; a version specific to Linux
E) Last Tested Software; the most recently tested version

2. What do you type to verify that Node.js was installed successfully?

A) `npm start`
B) `node install`
C) `javascript -v`
D) `node -v` or `node --version`
E) `check node`

3. Which package manager comes automatically bundled with Node.js when it is installed?

A) yarn
B) pnpm
C) bun
D) npm
E) pip

4. What is nvm (Node Version Manager) used for?

A) It provides a graphical interface for Node.js
B) It visualizes npm packages
C) It manages multiple Node.js versions on the same computer
D) It runs as a VS Code extension
E) It notifies you of Node.js updates

5. What is the extension of JavaScript source files?

A) `.java`
B) `.js.txt`
C) `.javascript`
D) `.jscript`
E) `.js`

6. What is the terminal command to run a `.js` file with Node.js?

A) `run merhaba.js`
B) `execute merhaba.js`
C) `start merhaba.js`
D) `npm merhaba.js`
E) `node merhaba.js`

7. Who developed Node.js and when?

A) Brendan Eich, 1995
B) Ryan Dahl, 2009
C) Tim Berners-Lee, 1990
D) Linus Torvalds, 2003
E) Douglas Crockford, 2008

8. What is the shortcut to create a new file in VS Code?

A) Cmd+N (Mac) / Ctrl+N (Windows)
B) Ctrl+S
C) Ctrl+O (Open)
D) Alt+F
E) Ctrl+P

9. What is the fundamental difference between the Node.js runtime environment and the browser JavaScript environment?

A) Node.js is slower
B) In Node.js there are no browser APIs like `window` or `document`; instead there are server APIs like `fs` and `http`
C) `const` / `let` cannot be used in the browser
D) Node.js only runs TypeScript
E) The browser uses more memory

10. After installation, the first `node -v` output shows `v24.1.0`. What does this mean?

A) Node.js major version 24, minor update 1, patch fix 0 is installed
B) 24 GB of memory is being used
C) The installation took 24 seconds
D) v24 is an experimental version and should not be used
E) Only version 24 is not LTS

▼ ANSWERS AND EXPLANATIONS

1 → C
LTS (Long-Term Support) is the stable version that provides long-term security and bug-fix support. It is recommended for production and learning environments. The Current version, on the other hand, tries out new features early but may be less stable.

2 → D
When you run the `node -v` or `node --version` command in the terminal, if you see a version number like `v24.x.x`, then Node.js has been installed successfully. If you get an error, it means the installation is not complete.

3 → D
npm (Node Package Manager) is installed automatically along with Node.js. You can check its version with `npm -v`. yarn, pnpm, and bun are alternative package managers and must be installed separately.

4 → C
nvm lets you install multiple Node.js versions on the same computer and switch between them easily. Different projects may require different Node versions; nvm solves this problem.

5 → E
JavaScript files use the `.js` extension. TypeScript may use `.ts`, and React components may use `.jsx` or `.tsx`. Forgetting the extension when saving the file name leads to not being able to run the code.

6 → E
The `node filename.js` command starts the Node.js runtime to run the specified JavaScript file. `npm start`, on the other hand, runs the `start` script in `package.json`, not a file directly.

7 → B
In 2009, Ryan Dahl separated Google's V8 JavaScript engine from the browser and ran it directly on the operating system. This made it possible for JavaScript to run on servers as well.

8 → A

`Cmd+N` (Mac) or `Ctrl+N` (Windows) opens a new empty file in VS Code. You can then save it with `Cmd+S` / `Ctrl+S` and give it an extension.

9 → B

In Node.js there are no browser APIs like `window`, `document`, or `localStorage`. Instead there are server APIs such as the file system (`fs`), networking (`http`), and process control (`process`). The JavaScript language is the same; the environment is different.

10 → A

Semantic versioning (SemVer) uses the `MAJOR.MINOR.PATCH` structure. `v24.11.0` → major version 24, minor 11, patch 0. Even major numbers (20, 22, 24) indicate LTS versions.

PART ZERO – FIRST STEPS

Chapter Z.4 – Writing Your First Program

● ○ ○ ○ ○ BEGINNER

> ◎ WHAT IS IT GOOD FOR?
>
> Until now you've been reading concepts. In this chapter it's **your** turn: you'll write, run, and see the result with your own eyes. Through small programs just a few lines long, we'll make the concepts of variables, expressions, and output concrete.

In 1972 at Bell Labs, Brian Kernighan wrote the smallest working program for the introductory documentation of the B programming language: `printf("hello, world\n")`. Over the following decades, this simple line became the first step of every programming book and every developer. Writing "Hello World" and seeing it run is part of a tradition.

Step by step: Age calculator

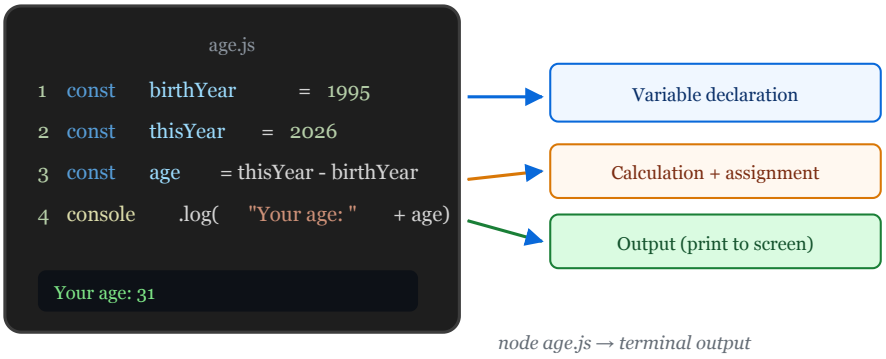


FIGURE Z.4.1 *Age calculator: 4 lines, 3 concepts (variable, operation, output)*

Open a new file in VS Code (`Cmd+N` / `Ctrl+N`). Name it `age.js` and save it into your `js-exercises` folder (`Cmd+S` / `Ctrl+S`).

Type the following code into it (I recommend typing it out by hand instead of copying; you're building muscle memory):

```js
const birthYear = 1995;
const thisYear = 2026;
const age = thisYear - birthYear;
console.log("Your age: " + age);
```

Run it from the terminal:

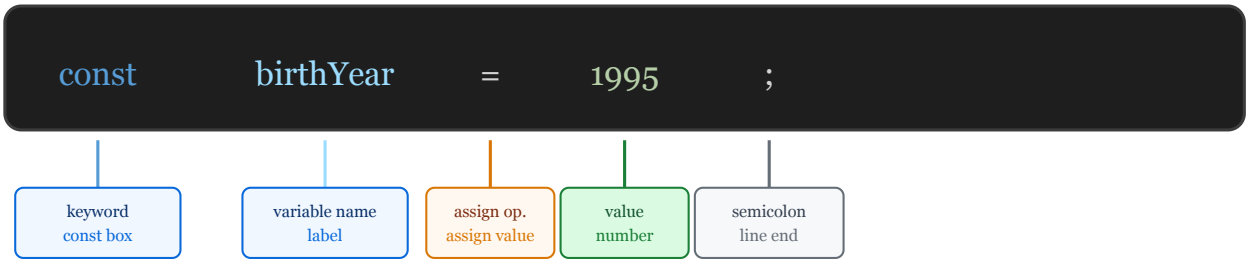
```
node age.js
```

Output:

```
Your age: 31
```

Try it: Replace the birth year with your own and save. Run it again from the terminal; did the result change? Then add a variable that holds your own name instead of `age` , and produce the output `"Hello, Your age: "` .

What does each line do?



The right side is evaluated first, then assigned to the name on the left

FIGURE Z.4.2 *Anatomy of a variable: const · name · = · value · ;*

In this little program, 4 separate concepts came together. Let's examine them one by one:

Lines 1–2: Variable declaration

```js
const birthYear = 1995;
const thisYear = 2026;
```

Anatomy:

PART	MEANING
`const`	"Create a constant variable"; its value cannot be changed later
`birthYear`	The variable's name (the name given by the developer)
`=`	Assignment operator ("assign this value")
`1995`	The value to assign (a number)
`;`	End of the line (not required but recommended)

In plain English: "*Create an unchangeable variable named `birthYear` and put the number 1995 into it.*"

Line 3: Calculation and Assignment

```js
const age = thisYear - birthYear;
```

The right side is evaluated first: `2026 - 1995 = 31`. Then the result is stored with the name `age`. This order matters: JavaScript always evaluates the expression first, then performs the assignment.

Line 4: Output

```js
console.log("Your age: " + age);
```

PART	MEANING
`console.log(...)`	Write what's in the parentheses to the screen
`"Your age: "`	Text (string); double quotes or single quotes don't matter
`+`	Combine two things (concatenation)
`age`	The variable holding the value `31`

Result: `"Your age: " + 31` → `"Your age: 31"` (the number is automatically converted to text; this is called **type coercion**; we'll explore it in detail later).

Quick test: what do you think `"1" + "1"` equals — `2` or `"11"`? The answer is `"11"`; when the `+` operator sees a string, it *concatenates*. But `"5" - 3` → `2`; subtraction always *coerces to a number*. This contradiction is one of JavaScript's most famous quirks — we'll explore these pitfalls in Chapter 2.

Comment Lines

When writing code, you want to leave notes answering "what does this line do?" These notes are written as **comments**. The computer **completely ignores comments**; they exist only for you or your team.

```js
// This is a single-line comment — it starts with //, not #
const age = 31; // you can also add comments to the right of a line

/*
  This is a multi-line comment.
  Use it for longer explanations or code you temporarily disable.
*/
const name = "Alex";
```

> • TIP
>
> Comments that answer "why did I write this code" are more valuable than "what does this code do". Better `// group messages per iteration` than `// increment counter`.

Concatenation with `+` works, but it strains the eyes and invites mistakes. Professionals do the same job more cleanly:

Modern alternative: Template literal

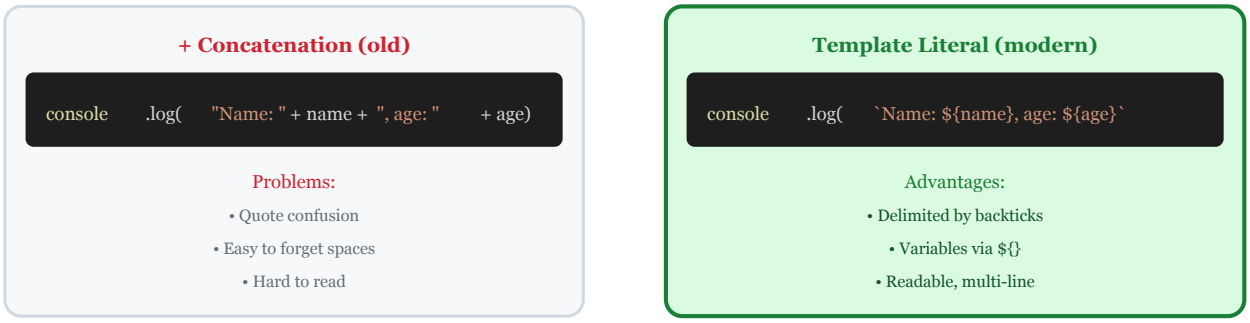


FIGURE Z.4.3 *String joining: + concatenation vs template literal*

Instead of joining with `+`, modern JavaScript uses **template literals** (template strings). Rather than quotes, they use backticks (`` ` ``) and the `${expression}` syntax:

```js
console.log(`Your age: ${age}`);
```

Their advantages:

- Adding multiple variables is easy: `` `${firstName} ${lastName}, ${age} years old` ``
- Line breaks are preserved (multi-line strings)
- Improves readability

Throughout this book, we'll prefer template literals. Turkish practice text — translating now.

Practice 1: Life Days Calculation

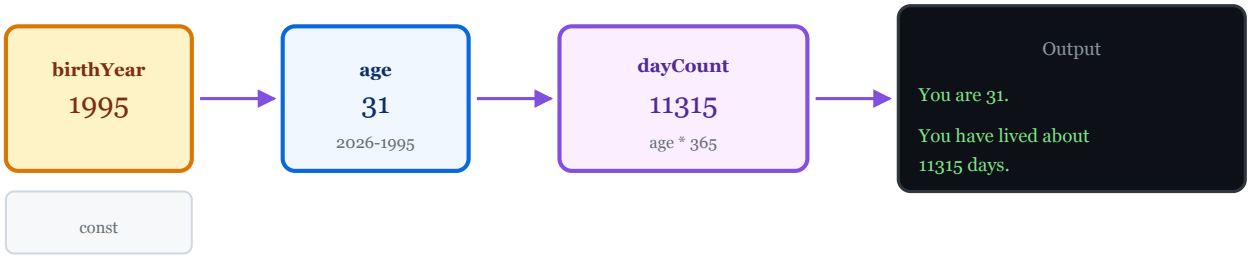


FIGURE Z.4.6 *Variable chain: computing from one to the next*

Try the following yourself:

1. Replace the birth year in the `age.js` file with **your own birth year**.
2. Add a new variable: `const dayCount = age * 365;`
3. Add a new `console.log`: `console.log(`You have lived about ${dayCount} days.`);`
4. Run it.

Expected output (for someone born in 1995):

```
You are 31.
You have lived about 11315 days.
```
OUTPUT

Ran into a problem? Read the error message carefully. It usually tells you the line number and reason for the error:

```
SyntaxError: missing ) after argument list
    at /path/to/age.js:4
```
ERROR

This message says "closing parenthesis missing on line 4." This is a normal part of development.

Practice 2: Unit Converter

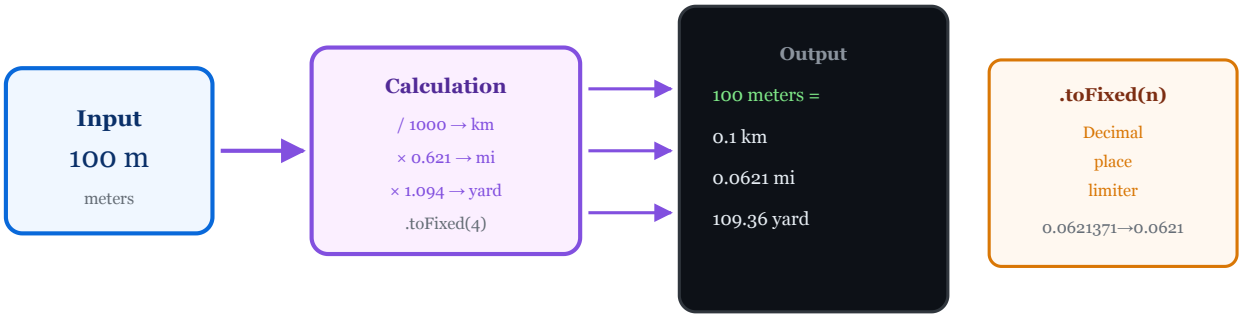


FIGURE Z.4.4 *Unit converter data flow: meters → kilometers, miles, yards*

Create a second file: `cevirici.js`. Write a program that calculates the following unit conversions:

```js
const metre = 100;
const kilometre = metre / 1000;
const mil = kilometre * 0.621371;
const yard = metre * 1.09361;

console.log(`${metre} meters =`);
console.log(`  ${kilometre} kilometers`);
console.log(`  ${mil.toFixed(4)} miles`);
console.log(`  ${yard.toFixed(2)} yards`);
```

`.toFixed(n)` limits the number of decimal places (`0.0621371` → `0.0621`).

Practice 3: Hour-second converter

```js
const saat = 3;
const saniye = saat * 60 * 60;
console.log(
  `${saat} hours = ${saniye.toLocaleString("tr-TR")} seconds`,
);
```

`toLocaleString("tr-TR")` displays the number in Turkish format (10.800).

EXERCISE Z.4 — Total price

Add the prices of three items. `pen = 5` , `notebook = 12` , `eraser = 3` . Store the total in a `total` variable, print it.

```
// Write your code here:
```

View solution → (Exercise Solutions section)

CHEAT SHEET — CHAPTER Z.4

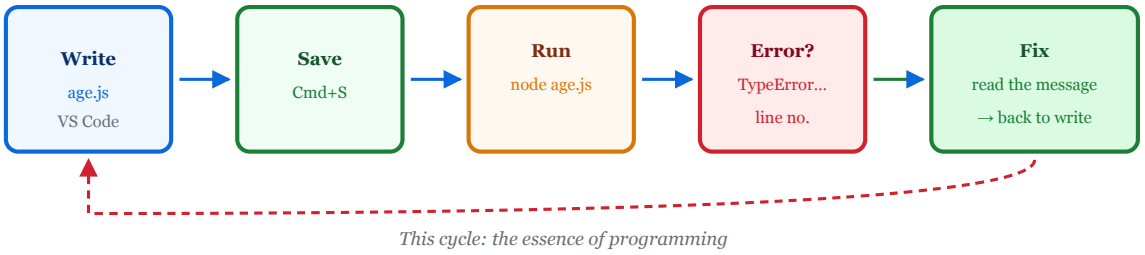


FIGURE Z.4.5 *The development cycle complete: write → save → run → error → fix*

TOPIC	EXAMPLE
Variable	`const name = "Alex";`
Print	`console.log(name);`
Math	`const age = 2026 - 1995;`
Concatenate	`` `Hello ${name}` ``

REVIEW QUESTIONS: CHAPTER Z.4

1. What does `//` do?
2. Explain what the statement `const age = 31;` means in plain English.

> • STRING OR NUMBER?
>
> In JavaScript, everything inside quotes is a **string**: `"3"`, `"hello"`, `"42"`. Numbers without quotes are a **number**: `3`, `42`, `3.14`. `"3 + 5"` is a string expression; `3 + 5` is a numeric addition operation.

3. Are `"3 + 5"` and `3 + 5` different? What's the output?
4. What does `console.log(1 + 1)` print?
5. What does `console.log("1" + "1")` print? (WARNING: trap)

▼ VIEW ANSWERS

Answers: 1) A comment; the computer ignores it. 2) "create a permanent box called age, put 31 in it". 3) `"3 + 5"` is a string, output `"3 + 5"`; `3 + 5` is math, output `8`. 4) `2`. 5) `"11"` (string concat).

You've written your first program; in the next chapter, we'll clarify the mental model for these variables and functions: variable = box, function = machine.

Chapter Z.4: Writing Your First Program – Quiz Questions

1. What does the following code print to the console when run?

```js
const birthYear = 1995;
const thisYear = 2026;
console.log("Your age: " + (thisYear - birthYear));
```

A) `Your age: 19952026`
B) `Your age: thisYear - birthYear`
C) `Your age: 31`
D) Throws an error
E) `31`

2. How do you write a single-line comment in JavaScript?

A) `/* comment */`
B) `// comment`
C) `<!-- comment -->`
D) `# comment`
E) `-- comment`

3. What is the result of the following expression?

```js
console.log("3" + 5);
```

A) `"35"`
B) `35`
C) `8`
D) Error
E) `3+5`

4. What is the result of the following expression?

```js
console.log("3" - 5);
```

A) `-2`
B) Error
C) `"3-5"`
D) `35`
E) `undefined`

5. What does the `console.log()` function do?

A) Deletes a variable
B) Assigns a value to a value
C) Terminates the program
D) Prints the value to the terminal (console)
E) Writes to a file

6. Who wrote the first "Hello World" program, and in which language's documentation did it appear?

A) Dennis Ritchie, C language
B) Guido van Rossum, Python
C) Brendan Eich, JavaScript
D) Bjarne Stroustrup, C++
E) Brian Kernighan, the B language

7. What is the keyboard shortcut for saving a file in VS Code?

A) Ctrl+Enter
B) Alt+S
C) Cmd+S (Mac) / Ctrl+S (Windows)
D) Cmd+W
E) Ctrl+X

8. How many variables are defined in the following code?

```js
const name = "Alex";
let age = 25;
const city = "Istanbul";
```

A) 1
B) 2
C) 0
D) 4
E) 3

9. The development loop is described as `Write → Save → Run → If there's an error, Read → Fix → Run again`. When does this loop end?

A) When the first error is fixed
B) When the Save step is done
C) When it's run 100 times
D) When the program produces the expected result and no errors remain
E) It never ends

10. Which of the following is true for a variable defined with `const`?

A) Its value can be changed later
B) Its value cannot be changed once defined
C) It can be defined a second time with the same name
D) It is function-scoped
E) It is hoisted and takes the value `undefined`

▼ ANSWERS AND EXPLANATIONS

1 → C
Inside the parentheses, `2026 - 1995 = 31` is computed, and the result is the number 31. Since one side of the `+` operator is a string (`"Your age: "`) and the other is a number (31), string concatenation occurs: `"Your age: 31"` .

2 → B
Everything from `//` to the end of the line is treated as a comment and ignored by the JavaScript engine. For multi-line comments, `/* ... */` is used. `#` denotes a comment in Python, `<!--` in HTML, and `--` in SQL.

3 → A
With the `+` operator, if one operand is a string, the other is converted to a string too. `"3"` is a string, `5` is a number → `"3" + "5"` → `"35"` . This is one of the most common gotchas in JavaScript.

4 → A
The `-` operator always performs a numeric operation. `"3"` is a string → it's converted to the number `3` → `3 - 5 = -2` . Unlike `+` , the `-` , `*` , `/` , and `%` operators never perform string concatenation.

5 → D
`console.log()` is JavaScript's most fundamental output function. It prints the given value (string, number, object, etc.) to the console. In Node.js it writes to the terminal, and in the browser it writes to the DevTools Console.

6 → E

In 1972, Brian Kernighan at Bell Labs wrote `printf("hello, world\n")` for the introductory documentation of the B programming language. This simple tradition later also appeared in the C language book and spread throughout the entire programming world.

7 → C

`Cmd+S` (Mac) / `Ctrl+S` (Windows) saves the active file. Unsaved files are marked with a dot (•) in the tab title in VS Code. Make it a habit to always save before running your code.

8 → E

Three variables are defined: `name`, `age`, and `city`. The `const` and `let` keywords specify the variable type; each declaration creates a separate variable.

9 → D

The loop is considered complete when the program runs correctly, produces the expected output, and no known errors remain. However, this loop repeats throughout your career, for every new feature or fix.

10 → B

A variable defined with `const` cannot be reassigned later (`const x = 5; x = 10;` → TypeError). However, the **contents** of an object or array defined with `const` can still be changed — this just means the reference itself doesn't change.

End of Sample

This is a **preview sample**.

This sample contains only the first chapters of *Code to Career: JavaScript & Node.js from Zero to Professional*. The full book spans 158+ chapters and is far more comprehensive, with exercises and solutions.

To get the full book, find *Code to Career* on Amazon KDP or Apple Books.

– Mert Türedü

From Zero to Professional.

One Path. No Gaps.

This book is your complete roadmap to master JavaScript and Node.js from the very first line of code to building real-world, production-ready applications.

You won't just learn the how, you'll understand the why. Every topic includes its history, deep explanations, visuals, exercises, and cheat sheets to make knowledge stick for life.

Whether you're a complete beginner or an experienced developer switching to JavaScript, this book adapts to you. Read it in order, or choose your own path.

JavaScript Foundations to Advanced Concepts

Variables, functions, closures, OOP, async programming, modules, ES features, and much more.

Node.js from Core to Real Projects

Event loop, streams, HTTP, REST APIs, authentication, databases, testing, and deployment.

Databases & SQL

PostgreSQL with real examples, queries, optimization, and best practices.

Modern Tools & Best Practices

ES Modules, TypeScript, testing, design patterns, security, performance, and more.

Hands-on Exercises & Cheat Sheets

Practice as you learn. Every chapter ends with exercises and quick reference notes.

Algorithmic Thinking for Interviews

Master problem solving with patterns, Big O, data structures, and classic interview questions.

MERT TÜREDÜ

Resources, updates and corrections:
mertturedu.thedev@gmail.com