

unknown · generics · keyof · infer · satisfies · Zod

THE COMPLETE PRACTICAL GUIDE

TypeScript from Zero to Production

From your first type annotation to type-safe systems in production

Node.js APIs · React front ends · Testing
Monorepos · Docker, CI/CD, and observability

TYPES · GENERICS · ZOD · FASTIFY · REACT

THENEXBOOKS

TypeScript from Zero to Production

The Complete Practical Guide
From your first type annotation to type-safe APIs,
React front ends, and production deployments

First Edition

THENEXBOOKS

TypeScript from Zero to Production
The Complete Practical Guide

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. TypeScript, Node.js, and the libraries discussed in this book evolve quickly; always consult the official documentation for the versions you use.

TypeScript, Visual Studio Code, and GitHub are trademarks of Microsoft Corporation. Node.js is a trademark of the OpenJS Foundation. JavaScript is a trademark or registered trademark of Oracle and/or its affiliates. React is a trademark of Meta Platforms, Inc. PostgreSQL, Docker, and other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is not affiliated with, sponsored by, or endorsed by any of these companies or projects.

PawPath, the running example in this book, is fictional, as are all clinics, pet owners, vets, pets, and payment providers mentioned in its examples. Any resemblance to real companies or products is coincidental.

Preface

JavaScript runs almost everywhere: in every browser, on servers, in serverless functions, in desktop and mobile apps, and in the build tools that produce all of them. Its flexibility is a gift when a program is small and a liability when it grows. A misspelled property, a value that is sometimes `undefined`, a function called with arguments in the wrong order — in plain JavaScript these mistakes surface at runtime, often in front of users. TypeScript moves a large class of them to the moment you type the code.

Getting started with TypeScript is easy: rename a file, add a few annotations, and the editor lights up with useful hints. Becoming productive with it on a real system is a different journey. You need to understand how the type checker reasons, when to lean on inference and when to be explicit, how to model a domain so that invalid states cannot be represented, how to validate data that arrives from the outside world, how to configure the compiler and the module system without fighting them, and how to keep a large codebase fast to type-check. Then you have to ship: a server, a front end, a background worker, tests, a CI pipeline, and a deployment that you can observe in production.

This book takes you along that whole journey. We start from zero — no prior TypeScript required — and finish with a production-ready system. Along the way we explain *why* each feature exists and where it breaks, not just *how* to type it.

Who This Book Is For

- **JavaScript developers** who want to adopt TypeScript properly rather than sprinkling `any` until the errors disappear.
- **Developers coming from typed languages** such as Java, C#, Go, or Kotlin who need to understand TypeScript's structural, flow-sensitive type system.
- **Full-stack and backend engineers** building Node.js APIs and React front ends who want types that span the whole stack.
- **Tech leads** who set conventions, migrate existing JavaScript codebases, and keep large TypeScript projects healthy.

What You Will Learn

- How the compiler thinks, how to set up a modern project, and the core of the type system: primitives, objects, functions, unions, and narrowing.
- Discriminated unions, classes, generics, mapped and conditional types, template literal types, utility types, branded types, and `satisfies`.
- Configuring `tsconfig` with confidence, ES modules versus CommonJS, and writing declaration files.

- Error handling, Result types, runtime validation with Zod, and asynchronous code with cancellation.
- Tooling: ESLint, Prettier, bundlers, and pnpm monorepos.
- Type-safe HTTP APIs with Fastify, typed SQL with Kysely, authentication and authorization, and end-to-end type safety.
- React with TypeScript, typed data fetching and forms, and testing with Vitest — including tests for your types.
- Scaling type-checking, migrating JavaScript incrementally, publishing libraries, and deploying with Docker, CI/CD, and observability.

How This Book Is Organized

Part I — Foundations (Chapters 1–5) explains what TypeScript is, sets up your tools, and covers the everyday type system through unions and narrowing. **Part II — The Type System in Depth** (Chapters 6–10) goes from classes and generics to type operators, conditional types, and type-level design. **Part III — Engineering with TypeScript** (Chapters 11–15) covers compiler configuration, modules, error handling and validation, asynchronous code, and tooling. **Part IV — TypeScript on the Server** (Chapters 16–20) builds the PawPath API, its data access, security, shared contracts, and background jobs. **Part V — Front End, Testing, and Scale** (Chapters 21–24) builds the React front end, tests everything, and keeps a growing codebase fast. **Part VI — Shipping to Production** (Chapters 25–28) covers migration, library publishing, deployment, and a complete case study. The appendices provide a cheat sheet, templates and a `tsconfig` reference, a production checklist, and a glossary.

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

The Running Example: PawPath

Throughout the book we build **PawPath**, a fictional network of veterinary clinics. Pet owners book appointments online, receptionists manage each day's schedule, vets record visits and vaccinations, the system issues invoices and takes payments, and a background worker sends vaccination reminders. That domain gives TypeScript plenty to do: appointments move through a state machine, many kinds of IDs must never be mixed up, money must be handled precisely, species-specific data calls for unions, and untrusted input arrives from forms and webhooks. PawPath is a pnpm monorepo with a shared package (`@pawpath/shared`), a Node.js API, a React web app, and a worker — so the same types flow from the database to the browser.

Versions and Conventions

Examples target recent TypeScript 5.x releases on a current Node.js LTS, using ES modules and `strict` mode. Where a feature arrived in a specific TypeScript release, we say so. The language and its ecosystem move quickly; where details may have changed since writing, a Note points you to the official documentation rather than guessing.

> **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 a behavior that can cause bugs, security problems, or outages in production.

Code appears in `monospace`. Long lines are wrapped to fit the page. Relative imports in Node.js code use `.js` extensions, as required by the `nodenext` module setting used throughout the book.

Contents

Preface

PART I — FOUNDATIONS

1 Why TypeScript: The Big Picture and How the Compiler Thinks

2 Setting Up: Node.js, the TypeScript Compiler, tsconfig, and Your Editor

3 Basic Types, Type Inference, and Annotations

4 Functions, Objects, Interfaces, and Type Aliases

5 Unions, Literal Types, Narrowing, and Discriminated Unions

PART II — THE TYPE SYSTEM IN DEPTH

6 Classes, Access Modifiers, and Object-Oriented Patterns

7 Generics from First Principles

8 Type Operators: keyof, typeof, Indexed Access, and Mapped Types

9 Conditional Types, infer, and Template Literal Types

10 Utility Types and Type-Level Design: Branded Types, satisfies, and as const

PART III — ENGINEERING WITH TYPESCRIPT

11 Mastering tsconfig: Strictness Flags, Targets, and Module Resolution

12 Modules, ESM and CommonJS, and Declaration Files

13 Error Handling, Result Types, and Runtime Validation with Zod

14 Asynchronous TypeScript: Promises, async/await, Iterators, and Cancellation

15 Tooling: ESLint, Prettier, Bundlers, and pnpm Monorepos

PART IV — TYPESCRIPT ON THE SERVER

16 Building a Type-Safe HTTP API with Node.js and Fastify

17 Typed Data Access: SQL, Kysely, Prisma, Drizzle, and Migrations

18 Authentication, Authorization, and Secure Input Handling

19 End-to-End Type Safety: Shared Packages, OpenAPI, and tRPC

20 Background Jobs, Events, Webhooks, and Integrations

PART V — FRONT END, TESTING, AND SCALE

21 TypeScript with React: Components, Props, Hooks, and Events

22 Typed Data Fetching, State Management, and Forms

23 Testing TypeScript: Vitest, Mocks, Type Tests, and Integration Tests

24 Performance at Scale: Type-Checking Speed, Project References, and Large Codebases

PART VI — SHIPPING TO PRODUCTION

25 Migrating a JavaScript Codebase to TypeScript Incrementally

26 Building and Publishing TypeScript Libraries

27 Deployment: Docker, CI/CD, Configuration, and Observability

28 Case Study: Taking PawPath to Production

APPENDICES

A TypeScript Cheat Sheet

B Templates and tsconfig Reference

C Production Checklist

D Glossary

PART I

Foundations

What TypeScript is and how its compiler reasons about your code, a modern project setup, and the everyday type system: primitives, inference, functions, objects, unions, narrowing, and the discriminated unions that model PawPath's appointments.

1 · Why TypeScript

2 · Setting Up

3 · Basic Types and Inference

4 · Functions, Objects, and Interfaces

5 · Unions and Narrowing

CHAPTER 1 · PART I — FOUNDATIONS

Why TypeScript: The Big Picture and How the Compiler Thinks

Before we write a single annotation, we need a clear mental model of what TypeScript is, what it does for you, and — just as important — what it never does. This chapter builds that model, walks through how the compiler turns your code into JavaScript, and introduces PawPath, the veterinary clinic platform we will build together across the book.

The Problem TypeScript Solves

JavaScript is a remarkable language. It runs in every browser, on servers through Node.js, in serverless functions, in mobile apps, and inside desktop applications. It is flexible, forgiving, and fast to write. Those same qualities become a liability as a codebase grows. When a project has ten files and one developer, you can hold the shape of every object in your head. When it has two thousand files, a dozen developers, and three years of history, nobody can.

Consider a small function from a clinic scheduling system written in plain JavaScript:

schedule.js

```
export function describeAppointment(appointment) {
  const start = new Date(appointment.startsAt);
  const end = new Date(start.getTime() + appointment.duration * 60_000);
  return `${appointment.pet.name} with ${appointment.vetName}, `
    + `${start.toLocaleTimeString()}–${end.toLocaleTimeString()}`;
}
```

Nothing about this function looks wrong. But ask a few questions and the uncertainty piles up. Is the field called `duration` or `durationMin`? Does the appointment object embed a `pet`, or does it only carry a `petId`? Is `vetName` always present, or only after the appointment is confirmed? Is `startsAt` a string, a number of milliseconds, or already a `Date`? The only ways to find out are to read every caller, search the codebase, run the code, or wait for a bug report. If the answer to the first question is `durationMin`, then `appointment.duration` is `undefined`, the multiplication produces `NaN`, and the function happily returns a string containing "Invalid Date". No exception is thrown. The bug simply travels downstream until a receptionist notices the schedule looks odd.

This is the core problem: in JavaScript, the *shape* of your data is an unwritten agreement between pieces of code. The agreement lives in developers' heads, in outdated comments, and in tests that may or may not cover the case. When the agreement changes — someone renames a field, makes a property

optional, or changes a number to a string — nothing tells you which other parts of the code relied on the old shape.

TypeScript makes that agreement explicit and checkable. Here is the same function with types:

schedule.ts

```
interface AppointmentSummary {
  startsAt: string; // ISO 8601, e.g. "2026-05-14T09:30:00Z"
  durationMin: number;
  petName: string;
  vetName: string;
}

export function describeAppointment(appointment: AppointmentSummary): string {
  const start = new Date(appointment.startsAt);
  const end = new Date(start.getTime() + appointment.duration * 60_000);
  //                                         ~~~~~~~~
  // error: Property 'duration' does not exist on type 'AppointmentSummary'.
  //        Did you mean 'durationMin'?
  return `${appointment.petName} with ${appointment.vetName}, `
    + `${start.toLocaleTimeString()}–${end.toLocaleTimeString()}`;
}
```

The mistake is now reported while you type, before the code ever runs, with a suggestion for the fix. More importantly, the interface becomes a single source of truth. If a teammate later renames `petName`, every place that reads the old name lights up with an error. The unwritten agreement is now written down, and a machine enforces it.

What you actually gain

Teams adopt TypeScript for several overlapping reasons, and it helps to name them precisely, because they shape how you use the language:

- **Early error detection.** Typos in property names, calling a function with the wrong number of arguments, forgetting that a value might be `undefined`, passing a string where a number is expected — whole categories of bugs are caught at compile time instead of in production.
- **Editor intelligence.** Because the editor knows the type of every expression, it can offer accurate autocompletion, show documentation on hover, jump to definitions, and find all references. This benefit is available even in files where you write very few annotations, because TypeScript infers most types for you.
- **Safe refactoring.** Renaming a symbol, changing a function signature, or splitting a module becomes a mechanical process: make the change, then follow the errors until there are none. In a large JavaScript codebase, the same change is a risky search-and-hope exercise.
- **Documentation that cannot rot.** A type signature such as `function scheduleVisit(petId: PetId, slot: TimeSlot): Promise<Appointment>` tells you what goes in and what comes out. Unlike a comment, it is checked on every build, so it cannot silently drift out of date.

- **Design pressure.** Writing down types forces you to decide what your data looks like — for example, whether a cancelled appointment must carry a reason.

Most of this value comes from ordinary types: object shapes, function parameters, and unions of a few strings. You can be productive long before you need the advanced features of Part II.

What TypeScript Is (and Is Not)

The one-sentence definition is: **TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.** Each part of that sentence matters.

A superset of JavaScript

Syntactically, TypeScript is JavaScript plus extra syntax for types. Every JavaScript feature — classes, modules, destructuring, async functions, optional chaining — is available with the same semantics. TypeScript does not invent a different way to write loops or a different object model. If you know modern JavaScript, you already know most of TypeScript's runtime behavior, because it *is* JavaScript's runtime behavior.

This superset relationship is why migration is possible at all. You can rename a `.js` file to `.ts`, and the code inside is still valid syntax. It may produce type errors — the checker might notice genuine problems or places where it cannot figure out what you meant — but the program itself does not need to be rewritten. Chapter 25 covers gradual migration of a real codebase in detail.

Typed

TypeScript adds a *static* type system. "Static" means the types are analyzed by reading the source code, without running it. The checker looks at every expression, figures out what type of value it can produce, and verifies that each value is used in a way that makes sense for its type. If you call `.toUpperCase()` on something that might be a number, you hear about it immediately.

The type system is optional in two senses. First, you do not have to annotate everything; inference fills in most types automatically. Second, you can opt out locally with escape hatches such as `any` or type assertions. These escape hatches are sometimes necessary, and we will discuss when they are acceptable, but every use weakens the guarantees the compiler can give you.

Compiles to plain JavaScript

Browsers and Node.js do not execute TypeScript's type system. What they run is JavaScript. The TypeScript compiler (the command-line tool is called `tsc`) reads your `.ts` files, checks them, and writes out `.js` files with all the type syntax removed. Other tools — esbuild, Vite, tsx, and recent versions of Node.js itself — can also strip type syntax, usually without checking the types at all. We examine these options in Chapter 2.

What TypeScript does not do

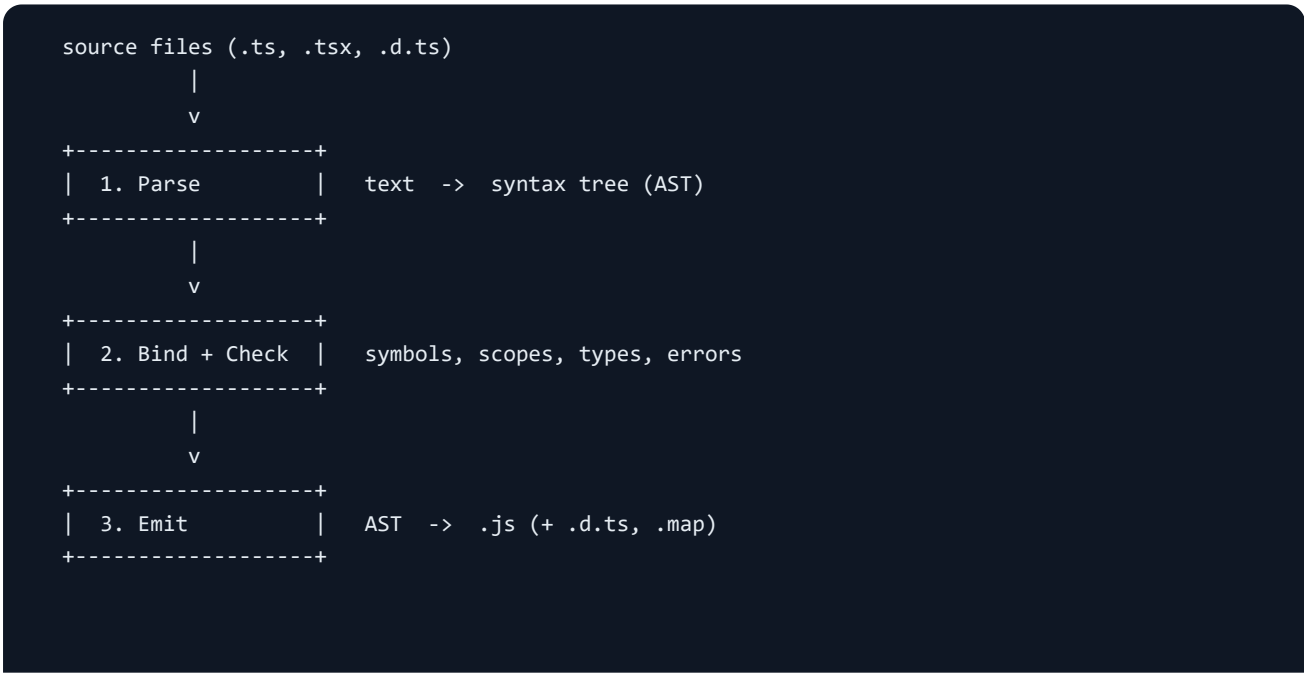
Understanding the limits is as important as understanding the features. TypeScript does **not**:

- Check data at runtime. If an HTTP request body claims to be a `Pet`, TypeScript has no way to verify that at runtime. The types describe what you *expect*; they do not inspect what actually arrives.
- Make your code faster. The emitted JavaScript runs at the same speed as hand-written JavaScript because it essentially is hand-written JavaScript with the annotations deleted.
- Change how JavaScript behaves. `0.1 + 0.2` is still not exactly `0.3`, `typeof null` is still `"object"`, and `this` still depends on how a function is called.
- Guarantee your program is correct. A program can be perfectly typed and still charge a pet owner twice. Types eliminate a class of mistakes; they do not replace tests, code review, or thinking.

> **WARNING**
>
> The most dangerous misconception about TypeScript is that a value typed as `Pet` is guaranteed to be a pet at runtime. It is only guaranteed if every path that produced the value was itself checked. Data from `JSON.parse`, `fetch`, environment variables, databases, message queues, and webhooks enters your program unchecked. If you simply assert its type, the compiler trusts you — and your bug surfaces far from its cause. Chapter 13 shows how to validate such data at the boundary with Zod.

How the Compiler Thinks: The Pipeline

A mental model of the compiler's work will make many later topics easier to understand, from confusing error messages to build performance. At a high level, the TypeScript compiler performs three stages:



```
          |
          v
    JavaScript output
```

Stage 1: Parsing

The parser reads each source file as text and builds an **abstract syntax tree** (AST) — a structured representation of the code in which a function declaration, its parameters, its body, and each expression inside become nodes. If the text is not syntactically valid (a missing brace, for example), the parser reports a syntax error. Parsing knows nothing about types; it only knows grammar.

Stage 2: Binding and type checking

The binder walks the AST and connects names to declarations: which `pet` does this line refer to, which scope is it in, which module exports `Money`? The result is a table of *symbols*. The checker then does the heavy lifting. For every expression it computes a type, and for every place a value flows — an assignment, a function argument, a return statement, a property access — it verifies that the source type is compatible with the target type. This is where the vast majority of errors you will see come from.

The checker computes types on demand, which is why your editor can ask for the type under the cursor without re-checking the whole project. Chapter 24 shows how to keep this process fast as a codebase grows.

Stage 3: Emitting

Finally, the emitter produces output. For most TypeScript syntax, emitting means **erasing**: deleting type annotations, interfaces, type aliases, and `import type` statements. Depending on your configuration, the emitter may also *downlevel* modern syntax to an older JavaScript version, convert module formats, generate declaration files (`.d.ts`) that describe your public types for other projects, and produce source maps that map the JavaScript back to your TypeScript for debugging.

A crucial and often surprising detail: **by default, type errors do not stop emission.** Unless you enable `noEmitOnError`, `tsc` will still write JavaScript files even when it reports type errors. The checker and the emitter are separate concerns. Emission only needs to know how to delete the types, which it can do regardless of whether those types are consistent.

> **NOTE**
>
> Microsoft is porting the TypeScript compiler to native code written in Go, announced as the basis for TypeScript 7. The goal is a much faster compiler and language service with the same type-checking behavior. The pipeline described here stays conceptually the same. Because the project is evolving, check the official TypeScript blog and repository for its current status before planning around it; Chapter 24 returns to the topic.

Type Erasure in Practice

Let us make erasure concrete. Here is a small TypeScript file from PawPath's shared package:

packages/shared/src/money.ts

```
export type Currency = "USD" | "EUR" | "GBP";

export interface Money {
  amountCents: number;
  currency: Currency;
}

export function addMoney(a: Money, b: Money): Money {
  if (a.currency !== b.currency) {
    throw new Error(`Cannot add ${a.currency} to ${b.currency}`);
  }
  return { amountCents: a.amountCents + b.amountCents, currency: a.currency };
}

export function formatMoney(m: Money, locale = "en-US"): string {
  return new Intl.NumberFormat(locale, {
    style: "currency",
    currency: m.currency,
  }).format(m.amountCents / 100);
}
```

And here is what the compiler emits, targeting a modern JavaScript version with ES modules:

packages/shared/dist/money.js

```
export function addMoney(a, b) {
    if (a.currency !== b.currency) {
        throw new Error(`Cannot add ${a.currency} to ${b.currency}`);
    }
    return { amountCents: a.amountCents + b.amountCents, currency: a.currency };
}
export function formatMoney(m, locale = "en-US") {
    return new Intl.NumberFormat(locale, {
        style: "currency",
        currency: m.currency,
    }).format(m.amountCents / 100);
}
```

The `Currency` type alias and the `Money` interface have vanished completely. There is no `Money` object at runtime, no class, no constructor. The parameter annotations are gone. The logic is byte-for-byte what you wrote, apart from formatting. This is type erasure: types exist only at compile time.

Erasure has consequences you need to internalize:

- **You cannot ask a type questions at runtime.** There is no way to write `if (value is Money)` and have TypeScript generate a check, because `Money` does not exist in the output. To check a shape at runtime, you write the check yourself or use a validation library that produces both a

runtime validator and a matching type.

- **Types add no runtime cost.** Adding more precise types never makes the program bigger or slower. You can model your domain as carefully as you like without worrying about bundle size.
- **The currency check inside `addMoney` still matters.** The `Currency` type guarantees each `currency` is one of three strings, but it cannot guarantee that two amounts share the same currency. That rule is enforced by the runtime `if` statement. Types and runtime checks complement each other.

The few features that are not erased

A handful of older TypeScript features generate runtime code rather than simply disappearing. The most common are `enum` declarations, `namespace` blocks that contain values, and *parameter properties* in class constructors (`constructor(private readonly repo: PetRepository)`). For example:

```
// TypeScript
enum Role {
  Owner = "owner",
  Receptionist = "receptionist",
}

// Emitted JavaScript (roughly)
var Role;
(function (Role) {
    Role["Owner"] = "owner";
    Role["Receptionist"] = "receptionist";
})(Role || (Role = {}));
```

These features predate the modern philosophy of "TypeScript is JavaScript plus erasable types". They still work and you will meet them in existing codebases, but they complicate tools that strip types without a full compiler. Recent Node.js versions, for instance, can run `.ts` files by stripping types, and that mode only supports erasable syntax. TypeScript 5.8 added the `--erasableSyntaxOnly` flag, which reports an error if you use one of these non-erasable constructs. In this book we generally prefer the erasable alternatives — union types instead of enums, ES modules instead of namespaces — and Chapter 3 and Chapter 10 explain why in more depth.

Structural Typing: Shapes, Not Names

If you come from Java, C#, or Swift, one aspect of TypeScript may feel strange: its type system is **structural**, not **nominal**. In a nominal system, a value is a `Pet` only if it was declared or constructed as a `Pet`. In a structural system, a value is compatible with `Pet` if it has the right *shape* — the required properties with compatible types — regardless of where it came from or what it was called.

structural.ts

```
interface Clinic {
  name: string;
  city: string;
}

interface Hotel {
  name: string;
  city: string;
}

function printLocation(clinic: Clinic): void {
  console.log(`${clinic.name} (${clinic.city})`);
}

const seaside: Hotel = { name: "Seaside Lodge", city: "Lisbon" };
printLocation(seaside); // OK: a Hotel has everything a Clinic needs

const fromApi = { name: "PawPath Porto", city: "Porto", phone: "+351 555 0100" };
printLocation(fromApi); // OK: extra properties are fine when passing a variable
```

`Clinic` and `Hotel` are different names, but TypeScript considers them interchangeable because their shapes match. The object stored in `fromApi` has an extra `phone` property, and that is fine too: a value with *more* properties than required still satisfies the type. Think of a type as a set of requirements, not as a label.

Structural typing exists because it matches how JavaScript is actually written: objects are created with literals, spread, merged, and passed around freely. A nominal system would force you to wrap every object in a class; structural typing describes existing JavaScript patterns without changing them.

Where structural typing bites

The flexibility has a cost. Consider PawPath's identifiers. A pet ID and an owner ID are both strings, so structurally they are identical:

```
function getPet(petId: string): Promise<Pet> { /* ... */ }

const ownerId = "own_3HX8PD";
await getPet(ownerId); // compiles fine — and fetches the wrong thing
```

The compiler cannot tell the two strings apart, because nothing in their types differs. In a system with half a dozen kinds of ID, mixing them up is a realistic bug. TypeScript offers a well-known technique for simulating nominal types, called **branded types**, which PawPath uses for every identifier:

```
type Brand<T, B extends string> = T & { readonly __brand: B };

type PetId = Brand<string, "PetId">;
type OwnerId = Brand<string, "OwnerId">;
```

```
declare function getPet(petId: PetId): Promise<Pet>;
declare const ownerId: OwnerId;

await getPet(ownerId);
// error: Argument of type 'OwnerId' is not assignable to parameter of type 'PetId'.
```

Do not worry about the syntax yet — generics appear in Chapter 7, and Chapter 10 explains brands fully, including how to create branded values safely. The point for now is that structural typing is the default, and when you need names to matter, you add a distinguishing property to the shape. The `__brand` property never exists at runtime; it is a purely compile-time marker, another example of erasure at work.

Excess property checks: a small exception

There is one place where TypeScript is deliberately stricter than pure structural typing. When you write an object literal directly where a specific type is expected, extra properties are reported as errors:

```
printLocation({ name: "PawPath Porto", city: "Porto", phnoe: "+351 555 0100" });
// error: Object literal may only specify known properties,
//        and 'phnoe' does not exist in type 'Clinic'.
```

The reasoning is practical. When you write a literal inline, an unknown property is almost always a typo or a misunderstanding of the API, and it would otherwise be silently ignored. When you pass a variable that was created elsewhere, the object might legitimately serve several purposes. Chapter 4 covers these excess property checks in detail, including how they interact with optional properties.

Soundness, Trade-offs, and the Escape Hatches

In type theory, a type system is *sound* if every program it accepts is guaranteed never to experience a type error at runtime. TypeScript is intentionally **not** fully sound. Its designers chose to accept some programs that could fail at runtime in exchange for being able to describe real-world JavaScript patterns conveniently. Knowing where the holes are helps you avoid falling into them.

unsound.ts

```
const weights: number[] = [4.2, 12.8, 31.0];
const fourth = weights[3];      // type: number (with default settings)
console.log(fourth.toFixed(1)); // runtime: TypeError, fourth is undefined

const raw: unknown = JSON.parse('{"name":"Biscuit"}');
const pet = raw as { name: string; species: string };
console.log(pet.species.toUpperCase()); // runtime: TypeError, species is undefined

const anything: any = "not a number";
const n: number = anything;     // no error: any turns checking off
console.log(n.toFixed(2));      // runtime: TypeError
```

Each example compiles without errors, yet each crashes at runtime. The first shows that array indexing is assumed to succeed by default; the `noUncheckedIndexedAccess` option, covered in Chapter 11, changes the type of `weights[3]` to `number | undefined`. The second shows a **type assertion** (`as`), which tells the compiler to trust you. The third shows `any`, which switches off checking for everything it touches and spreads silently through assignments.

These are not flaws to be outraged about; they are deliberate trade-offs. But they suggest a working philosophy that we follow throughout this book:

1. Turn on `strict` mode from day one. It enables a family of checks, including `strictNullChecks`, without which `null` and `undefined` silently belong to every type.
2. Treat `any` as a last resort, and prefer `unknown` when you genuinely do not know a type. Chapter 3 shows the difference.
3. Treat `as` assertions as claims you must be able to defend. Every assertion is a spot where you have taken over the compiler's job.
4. Validate untrusted data at runtime at the edges of your system, so the types inside the system reflect reality.

> **TIP**
> A useful habit: when you review a pull request in a TypeScript codebase, search the diff for `any`, `as`, `!` (the non-null assertion), and `@ts-ignore`. These are the places where a human has overruled the compiler. They are not always wrong, but each deserves a sentence of justification.

Gradual Typing and Inference

TypeScript is designed around **inference**: the compiler works out types from how values are created and used, so you only annotate where it adds information. Look at this snippet:

```
const clinicName = "PawPath Lisbon";        // type: "PawPath Lisbon"
let visitCount = 0;                         // type: number
const species = ["dog", "cat", "rabbit"];   // type: string[]
const pet = { name: "Biscuit", ageYears: 4 };
// type: { name: string; ageYears: number }

function nextCount(current: number) {
  return current + 1;                       // return type inferred: number
}
```

Only one annotation appears — on the function parameter, where the compiler has no other way to know what callers will pass. Everything else is inferred. This is typical of well-written TypeScript: annotations at boundaries (function parameters, exported APIs, public data structures) and inference inside function bodies. Chapter 3 goes deep on inference, including why `clinicName` gets the literal type `"PawPath Lisbon"` while `visitCount` gets the general type `number`.

Where TypeScript Runs: One Language, Many Targets

Because TypeScript compiles to JavaScript, it can go anywhere JavaScript goes. In practice, you will use it in several environments, each with its own toolchain:

Environment	Typical tooling	Covered in
Node.js servers	tsc, tsx, Node type stripping	Chapters 2, 16–20
Browser apps	Vite (esbuild/Rollup), React	Chapters 21–22
Shared libraries	tsc, tsup, declaration files	Chapters 12, 26
Monorepos	pnpm workspaces, project references	Chapters 15, 24
Containers and CI	Docker, type-check in CI	Chapter 27

A recurring theme is the separation of two jobs: **type checking** and **transpiling**. Fast tools like esbuild, Vite, and tsx transpile each file independently by stripping types, which is quick but skips checking entirely. The TypeScript compiler performs the full check. Production setups typically run both: a fast transpiler for development and bundling, and `tsc --noEmit` in the editor and in continuous integration to verify types. You will see this pattern repeatedly, starting in Chapter 2.

There is also an early-stage TC39 **proposal**, often called "type annotations", that would let JavaScript engines parse and ignore type-like syntax. Its future is uncertain; for everything in this book, TypeScript needs a tool to strip its types before an engine can run it.

A Tour of PawPath

Learning a type system from isolated snippets only goes so far. Types earn their keep in real systems with real messiness: data from forms, rules that depend on state, the same model shared by several programs. So throughout the book we build **PawPath**, a platform for a fictional network of veterinary clinics.

What PawPath does

- **Pet owners** register their pets and book appointments online at a clinic of their choice.
- **Receptionists** manage each clinic's daily schedule: confirming requests, checking pets in, handling cancellations and no-shows.
- **Vets** record visits, diagnoses, prescriptions, and vaccinations.
- The system issues **invoices** and takes payments through Paylark, a fictional payment provider that notifies us through webhooks.
- A **background worker** sends vaccination reminders and invoice emails.

Every one of these features stresses a different part of TypeScript, which is exactly why we chose this domain.