Learn TypeScript from Zero to Hero: A Complete Beginner-to-Advanced Guide

December 13, 2025 · 13 min · 2661 words · martinuke0

Introduction

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It brings static typing, modern language features, and first-class tooling to the world's most popular programming language. Whether you're building front-end apps, Node.js services, or publishing libraries, TypeScript helps you catch bugs earlier, refactor safely, and communicate intent through types.

This guide takes you from zero to hero. We'll start with the essentials and build up to advanced topics like generics, conditional types, module augmentation, project references, and publishing typed libraries. You'll see practical examples, configuration tips, and real-world best practices. At the end, you'll find a curated list of high-quality resources.

> Note
>
> - You can adopt TypeScript incrementally—file by file or even via JSDoc types in .js files.
> - Focus on understanding type inference and narrow types; you don't need to annotate everything.

ble of contents

Getting started

1. Install TypeScript and a runtime tool:

```
npm i -D typescript
npm i -D tsx # fast TS runner for dev (alternative: ts-node)
npx tsc --init
```

2. Create a tsconfig.json:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "moduleResolution": "Bundler",
    "strict": true,
    "noUncheckedIndexedAccess": true,
    "exactOptionalPropertyTypes": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    dModules": true,
    ": ".",
```

```
      "paths": {
        "@/*": ["src/*"]
      },
      "outDir": "dist"
    },
    "include": ["src"]
  }
```

3. Add scripts:

```
{
  "scripts": {
    "build": "tsc -p tsconfig.json",
    "dev": "tsx watch src/index.ts",
    "start": "node dist/index.js",
    "typecheck": "tsc -p tsconfig.json --noEmit"
  }
}
```

4. Editor setup:
   - Use VS Code with the official TypeScript extension (bundled).
   - Enable “Use Workspace Version” for TS in VS Code to match your project.

> Tip Start with “strict”: true. It’s easier to relax a few flags later than to retrofit strictness.

TypeScript basics

Primitives, arrays, tuples

```
let id: number = 42;
let name: string = "Ada";
let active: boolean = true;

const tags: string[] = ["ts", "js"];
const counts: Array<number> = [1, 2, 3]; // alternative syntax

// Tuples: fixed length and positions
const point: [x: number, y: number] = [10, 20];
'' Named tuples (better readability)
  e Point = [x: number, y: number];
```

wn, never, void

- any: opt-out of type checking. Avoid; it defeats TypeScript's benefits.
- unknown: like any but must be narrowed before use (safer).
- never: represents impossible values (e.g., function that always throws).
- void: absence of a value (e.g., function returning nothing).

```
function parse(input: unknown) {
  if (typeof input === "string") return JSON.parse(input);
  throw new Error("invalid input");
}

function fail(message: string): never {
  throw new Error(message);
}

function log(msg: string): void {
  console.log(msg);
}
```

Unions, intersections, literals

```
type Status = "idle" | "loading" | "success" | "error"; // literal union

type ApiResult =
  | { ok: true; data: unknown }
  | { ok: false; error: string }; // discriminated union

type WithTimestamps = { createdAt: Date; updatedAt: Date };
type User = { id: string; name: string } & WithTimestamps; // intersection
```

Interfaces vs type aliases

- interface: extendable declarations; supports declaration merging.
- type: more flexible (unions, intersections, mapped/conditional types).
- Both describe shapes; use what fits. Many teams prefer type for consistency and interface for public APIs.

```
interface Person {
  id: string;
  name: string;
}

= Person["id"];
```

Feedback

```
type Response<T> = { ok: true; data: T } | { ok: false; error: string };

// Interface extension
interface Admin extends Person {
  role: "admin";
}
```

Functions

Parameters, returns, overloads

```
// Parameter and return type annotations
function add(a: number, b: number): number {
  return a + b;
}

// Optional and default params
function greet(name = "world"): string {
  return `Hello, ${name}`;
}
function format(id: string, prefix?: string): string {
  return `${prefix ?? "ID"}:${id}`;
}

// Rest params
function sum(...nums: number[]): number {
  return nums.reduce((a, b) => a + b, 0);
}

// Overloads for better call-site typing
function toArray(x: string): string[];
function toArray<T>(x: T): T[];
function toArray<T>(x: T): T[] {
  return [x];
}
```

This and call signatures

```
interface ClickHandler {
  (event: MouseEvent): void; // call signature

const obj = {

is: { count: number }, delta: number) {
```

```
    this.count += delta;
  }
};
obj.increment(1);
```

> Note Avoid using this in standalone functions. Prefer explicit params.

Objects, interfaces, and classes

Index signatures and readonly

```
// Index signatures for dynamic keys
type Dictionary<T> = {
  [key: string]: T;
};

type ReadonlyPoint = {
  readonly x: number;
  readonly y: number;
};

const settings: Dictionary<string> = {};
settings.theme = "dark";
```

Classes, access modifiers, abstract

```
class Queue<T> {
  #items: T[] = []; // ECMAScript private field

  enqueue(item: T) {
    this.#items.push(item);
  }
  dequeue(): T | undefined {
    return this.#items.shift();
  }
  get size(): number {
    return this.#items.length;
  }
}

tract class Shape {
  constructor(public color: string) {}
  abstract area(): number;
```

```
class Rectangle extends Shape {
  constructor(public w: number, public h: number, color: string) {
    super(color);
  }
  area() {
    return this.w * this.h;
  }
}

// Implements for structural contracts
interface Serializable {
  toJSON(): string;
}

class UserModel implements Serializable {
  constructor(public id: string, public name: string) {}
  toJSON() {
    return JSON.stringify({ id: this.id, name: this.name });
  }
}
```

> Tip Prefer ECMAScript private fields (#field) for runtime privacy. TS's private modifier is type-only privacy.

Generics

Generics let you write reusable, type-safe components.

```
function first<T>(items: T[]): T | undefined {
  return items[0];
}

function prop<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key];
}
```

Constraints and defaults

```
type WithId = { id: string };

function byId<T extends WithId>(items: T[], id: string): T | undefined {
  return items.find(i => i.id === id);
}

interface ApiOptions<T = unknown> {
```

```
  parse?: (raw: unknown) => T;
}
```

Generic classes and interfaces

```
interface Repository<T, Id = string> {
  get(id: Id): Promise<T | null>;
  set(id: Id, value: T): Promise<void>;
}

class MemoryRepository<T, Id = string> implements Repository<T, Id> {
  private store = new Map<Id, T>();
  async get(id: Id) { return this.store.get(id) ?? null; }
  async set(id: Id, value: T) { this.store.set(id, value); }
}
```

Narrowing and control flow

TypeScript narrows types based on checks—making unions safe to use.

```
function lengthOf(x: string | string[]): number {
  if (typeof x === "string") return x.length; // x: string
  return x.length; // x: string[]
}
```

Type guards and predicates

```
type Cat = { type: "cat"; meow(): void };
type Dog = { type: "dog"; bark(): void };
type Pet = Cat | Dog;

function isCat(pet: Pet): pet is Cat {
  return pet.type === "cat";
}

function speak(pet: Pet) {
  if (isCat(pet)) pet.meow();
  else pet.bark();
}
```

Assertion functions and exhaustive checks

```
function assertDefined<T>(value: T): asserts value is NonNullable<T> {
```

```
  if (value == null) throw new Error("Expected value to be defined");
}

type Status = "idle" | "loading" | "success" | "error";

function handleStatus(s: Status) {
  switch (s) {
    case "idle":
    case "loading":
    case "success":
    case "error":
      break;
    default: {
      const _exhaustive: never = s; // helps catch future cases
      return _exhaustive;
    }
  }
}
```

> Tip Prefer narrowing early and often. It trims unsafe branches and clarifies logic.

Advanced types

keyof, indexed access, mapped types

```
type Person = { id: string; name: string; age?: number };

type PersonKeys = keyof Person;           // "id" | "name" | "age"
type NameType = Person["name"];           // string
type OptionalAge = Person["age"];         // number | undefined

type PartialRecord<K extends PropertyKey, T> = {
  [P in K]?: T;
};

type ReadonlyPerson = {
  readonly [K in keyof Person]: Person[K];
};
```

Conditional types and infer

```
type Awaited<T> = T extends Promise<infer U> ? U : T;

type ElementType<T> = T extends (infer U)[] ? U : never;
```

```
type IfElse<C extends boolean, T, F> = C extends true ? T : F;
```

Distributive conditional types apply over unions. You can prevent distribution with brackets:

```
type ToArray<T> = T[]; // A | B -> (A|B)[]
type ToArrayDistribute<T> = T extends any ? T[] : never; // A | B -> A[] | B[]
type NoDistribute<T> = [T] extends [any] ? T[] : never;   // prevent distribution
```

Template literal types and utility types

```
type EventName<E extends string> = `on${Capitalize<E>}`;
type ButtonEvents = EventName<"click" | "focus">; // "onClick" | "onFocus"

// Built-in utilities
type PartialPerson = Partial<Person>;
type RequiredPerson = Required<Person>;
type ReadonlyPerson2 = Readonly<Person>;
type Picked = Pick<Person, "id" | "name">;
type Omitted = Omit<Person, "age">;
type RecordMap = Record<string, number>;
```

> Tip Use the satisfies operator to validate structure without widening:
>
> ```
> const config = {
>   mode: "prod",
>   retries: 3
> } as const satisfies { mode: "prod" | "dev"; retries: number };
> ```

Modules and project structure

ESM vs CommonJS

Modern projects should prefer ES modules.

```
// ESM
import { readFile } from "node:fs/promises";
  port function load(path: string) { return readFile(path, "utf8"); }

// CJS
  adFile } = require("node:fs/promises");
  ts = { load };
```

Feedback

tsconfig tips:

- “module”: “ESNext” or “NodeNext” for Node ESM.
- “moduleResolution”: “Bundler” (for bundlers) or “NodeNext” (Node’s resolver).

Path mapping and barrel files

Use baseUrl and paths for cleaner imports, but keep runtime aware via bundler or tsconfig-paths.

```
{
  "compilerOptions": {
    "baseUrl": ".",
    "paths": { "@/*": ["src/*"] }
  }
}
```

> Caution Barrel files (index.ts re-exporting many modules) can create circular dependencies. Monitor build output and import graphs.

Asynchronous TypeScript

Promises and async/await

```
async function fetchJson<T>(url: string): Promise<T> {
  const res = await fetch(url);
  if (!res.ok) throw new Error(`HTTP ${res.status}`);
  return (await res.json()) as T;
}
```

Typing fetch and APIs

Prefer explicit response types and validation at boundaries.

```
type User = { id: number; name: string };

nc function getUser(id: number) {
eturn fetchJson<User>(`/api/users/${id}`);
}
```

> Tip Type assertions don't validate data. Consider runtime validators (e.g., Zod, valibot) to verify JSON shapes.

DOM and Node typings

- Target DOM types by ensuring "lib": ["ES2020", "DOM"] (default in many setups).
- For Node.js, install @types/node and set "moduleResolution": "NodeNext" or "Bundler".

```
npm i -D @types/node
```

```
{
  "compilerOptions": {
    "types": ["node"]
  }
}
```

Decorators (modern)

TypeScript 5+ supports the standard ECMAScript decorators proposal. Decorators let you annotate classes, methods, and fields.

```
function log() {
  return function<T extends Function>(value: T, context: ClassMethodDecoratorContext)
    return function(this: unknown, ...args: unknown[]) {
      console.log(`Called ${String(context.name)} with`, args);
      // @ts-ignore - value is a function
      return value.apply(this, args);
    } as unknown as T;
  };
}

class Greeter {
  @log()
  greet(name: string) {
    return `Hello, ${name}`;
  `
```

> - Ensure your TypeScript version supports standard decorators (TS 5+). Check the TypeScript release notes and your tsconfig.
> - Legacy decorators (experimentalDecorators) use a different API signature; avoid mixing styles.

Working with third-party libraries

DefinitelyTyped and @types

Many JS libraries ship with types. If not, install community-maintained types:

```
npm i -D @types/lodash
```

TypeScript automatically picks them up.

Module augmentation

You can add types to existing modules.

```
// types/lodash.overrides.d.ts
import "lodash";
declare module "lodash" {
  interface LoDashStatic {
    shout(s: string): string;
  }
}

import _ from "lodash";
_.mixin({ shout: (s: string) => s.toUpperCase() + "!" });
_.shout("hi"); // typed
```

You can also augment the global scope:

```
// global.d.ts
declare global {
  interface Window {
    myAppVersion: string;

export {};
```

tsconfig deep dive

Key options for robust projects:

- strict: enables strict type-checking family; keep it on.
- strictNullChecks: null/undefined must be handled explicitly.
- exactOptionalPropertyTypes: optional props aren't automatically | undefined for assignment—more precise.
- noUncheckedIndexedAccess: index access returns T | undefined, preventing runtime misses.
- noImplicitAny: flags implicit any.
- isolatedModules: each file is transpiled in isolation; required by many bundlers.
- moduleResolution: "Bundler", "Node", "NodeNext"—match your runtime.
- target/module: choose appropriate ECMAScript versions for output.
- lib: set JS environment libs (e.g., ["ES2022", "DOM"]).
- declaration/declarationMap: emit .d.ts and their source maps for libraries.
- sourceMap: useful for debugging.
- incremental/composite: speed up builds and enable project references.
- baseUrl/paths: configure path aliases.

Example for a library:

```
{
  "compilerOptions": {
    "target": "ES2020",
    "module": "ESNext",
    "declaration": true,
    "declarationMap": true,
    "outDir": "dist",
    "strict": true,
    "skipLibCheck": true,
    "composite": true
  },
  "include": ["src"]
```


with TypeScript

You can type-check tests and run them with modern test runners.

Example with Vitest:

```
npm i -D vitest tsx @types/node
```

```
{
  "scripts": {
    "test": "vitest",
    "test:watch": "vitest watch",
    "typecheck": "tsc --noEmit"
  }
}
```

A simple typed test:

```
// src/math.test.ts
import { describe, it, expect } from "vitest";
import { add } from "./math";

describe("add", () => {
  it("adds numbers", () => {
    expect(add(2, 3)).toBe(5);
  });
});
```

Jest works too (via ts-jest or Babel). Ensure tsconfig aligns with your test environment (e.g., DOM for front-end, node for back-end).

Tooling and builds

- tsc: the TypeScript compiler—type checks and emits JS.
- tsx or ts-node: run TS directly in dev.
- Bundlers: Vite, esbuild, tsup, Rollup, Webpack.

Example tsup build for libraries:

```
i -D tsup
```

```
{
  "build": "tsup src/index.ts --format esm,cjs --dts"
```

```
  }
}
```

> Tip Separate type checking from bundling for speed. Many bundlers transpile TS without type-checking—run tsc –noEmit in CI.

Migrating from JavaScript to TypeScript

- Start with JSDoc types and “checkJs”: true for gradual adoption.
- Enable allowJs to compile .js alongside .ts.
- Convert leaf modules first, then core modules.
- Turn on strict mode early; temporarily use any where needed but chip away at it.

```
{
  "compilerOptions": {
    "allowJs": true,
    "checkJs": true,
    "strict": true
  },
  "include": ["src"]
}
```

JSDoc example:

```
/**
 * @param {number[]} nums
 * @returns {number}
 */
export function average(nums) {
  return nums.reduce((a, b) => a + b, 0) / nums.length;
}
```

Publishing typed libraries

1. Emit declarations:
   - tsconfig: “declaration”: true, “declarationMap”: true.
2. Package.json fields:

```
  "name": "my-lib",
  "type": "module",
  "exports": {
    ".": {
      "types": "./dist/index.d.ts",
      "import": "./dist/index.js",
      "require": "./dist/index.cjs"
    }
  },
  "types": "./dist/index.d.ts",
  "typesVersions": { "*": { "*": ["dist/*"] } }
}
```

3. Consider dual ESM/CJS builds if users still need CJS.
4. Include a tsconfig.build.json for clean builds.
5. Test your types with dtslint or by consuming your package in a sample repo.

> Tip Avoid exporting types that reference private/internal modules. Keep your public API surface small and stable.

Best practices and common pitfalls

- Prefer type inference. Annotate public APIs, complex generics, and boundary points (I/O).
- Use unknown instead of any at boundaries; validate at runtime.
- Prefer unions of string literals over enums for flexibility; if you need a runtime object, use const assertions:

```
const Status = { Idle: "idle", Loading: "loading" } as const;
type Status = (typeof Status)[keyof typeof Status];
```

- Use the satisfies operator to ensure object shapes without widening.
- Avoid ambient global types in apps; keep declarations local to modules unless necessary.
- Narrow early; use discriminated unions and exhaustive checks.
- Don’t overuse type assertions (as). If you assert, do it close to the source and stify with a comment.
- Keep tsconfig strict. Resist disabling strictNullChecks or alPropertyTypes.

- Lint with typescript-eslint for consistency and additional correctness rules.

Performance and scaling tips

- Enable incremental builds and use project references for monorepos.
- Use skipLibCheck to speed up type-checking (acceptable in most apps).
- Split large types into named helpers; heavy conditional types can slow the checker.
- Avoid deep wildcard re-exports; import directly where possible.
- Cache node_modules in CI and run tsc –noEmit separately from bundling.

Conclusion

TypeScript elevates JavaScript development with expressive, static types and excellent tooling. By mastering fundamentals like unions, generics, and narrowing—and layering in advanced patterns like conditional/mapped types, module augmentation, and robust tsconfig settings—you'll build safer, more maintainable software.

Adopt TypeScript incrementally, type your boundaries, and let inference carry the rest. As your codebase grows, TypeScript becomes an indispensable ally: catching bugs early, guiding refactors, and documenting intent. Use the resources below to continue your journey and stay current with new language features.

Resources

- Official docs and handbook
  - TypeScript Docs: https://www.typescriptlang.org/docs/
  - Handbook (Reference): https://www.typescriptlang.org/docs/handbook/intro.html
  - TSConfig Reference: https://www.typescriptlang.org/tsconfig
  - Release Notes: https://www.typescriptlang.org/docs/handbook/release-
    rview.html

- Deep dives and guides
  - TypeScript Deep Dive (Basarat): https://basarat.gitbook.io/typescript/
  - Effective Type

TypeScript | Javascript | Web-Development | Programming | Best Practices

« PREV
Database Indexes Internally: A Deep Dive into Data Structures and Operations

NEXT »
How to Start a Nonprofit Organization in Any Country: A Step-by-Step Global Playbook