Module 13 of 16

TypeScript preview

Erased types and inference; any, unknown and never; interfaces, structural typing and discriminated unions; generics, keyof and the utility types; narrowing, type guards and assertions; tsconfig, declaration files, builds and migration.

TypeScript is the JavaScript most teams write now: the same language with a static type system layered on top, checked at build time and erased before the code runs. The type system is worth learning as its own subject — inference and widening, structural assignability, discriminated unions, generics constrained by `keyof`, narrowing by control flow — because those rules are what the compiler applies to every line, and understanding them turns error messages from noise into instructions.

This module is a preview, not a full course, and it is judged as JavaScript: Node runs the exercises, so each one models a piece of the type system at run time — a function that infers what `const` / `let` / `as const` would give, a structural assignability checker with the excess-property rule, utility types as operations on schemas, a narrowing engine over a union, guards whose bodies are the runtime checks the types rely on. The lessons cover the basics and the three special types; object types, interfaces and unions; functions, generics and utility types; narrowing and guards; and the practical layer of tsconfig, declaration files, builds and migration.

The checkpoint builds a schema library that validates values and prints the type it would infer, an exhaustive state machine over discriminated-union states and events, and a declaration-file generator — three tasks that show the type system and the runtime meeting in the middle.

CONTENTS

1	What TypeScript adds — erased types, inference, and any versus unknown versus never	13 min
2	Object types, interfaces, structural typing and discriminated unions	13 min
3	Function types, generics, keyof and the utility types	13 min
4	Narrowing, type guards, assertions — making the compiler trust a value	12 min
5	TypeScript in practice — tsconfig, declaration files, builds, migration and the errors you will see	12 min
✓	Checkpoint — TypeScript preview	24 min

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

LESSON 1 · 13 MIN

What TypeScript adds — erased types, inference, and any versus unknown versus never

TypeScript is JavaScript plus a **static type system**: annotations and inference that let a compiler check your program before it runs, then erase everything and emit plain JavaScript. Nothing about the runtime changes — no types exist at run time, no performance difference, the same engines — which is both the point and the trap: a type is a *promise about* a value, checked at compile time, and a value that arrives from JSON, a database or a user can break the promise without anyone noticing unless you check at the boundary. This lesson covers what the compiler does, the basic types and how inference assigns them, literal types and widening, and the three special types — `any`, `unknown`, `never` — whose correct use is most of the difference between TypeScript that helps and TypeScript that lies.

Erasure, and what that means

```
function total(items: { price: number; qty: number }[]): number {
  return items.reduce((sum, i) ⇒ sum + i.price * i.qty, 0);
}
```

compiles to

```
function total(items) {
  return items.reduce((sum, i) ⇒ sum + i.price * i.qty, 0);
}
```

`tsc` (or esbuild, swc, Babel — anything that strips types) removes the annotations. Consequences: a type error is a compile-time message, never a runtime exception; `instanceof SomeInterface` cannot exist; a value's runtime shape is whatever it actually is, whatever the annotation claimed. TypeScript's guarantee is conditional: *if* every value entering the program matches its declared type, *then* the program has no type errors. Making the "if" true at the boundaries (lesson 4) is your job.

Basic types and inference

```
let n: number = 3;            // annotation
let s = "hi";                 // inferred: string
let ok = true;                // boolean
let big = 10n;                // bigint
let nothing: null = null;     // null and undefined are types too
let ids: number[] = [];       // array of numbers — also Array<number>
let pair: [string, number] = ["a", 1];   // tuple: fixed length and per-position t
let maybe: string | undefined;           // union — one of several types
const point = { x: 1, y: 2 };            // inferred { x: number; y: number }
```

Inference assigns types from initialisers and return statements; annotate parameters (nothing to infer from), public function return types when you want the contract explicit, and variables declared without an initialiser. Over-annotating (`const n: number = 3`) adds noise; under-annotating parameters gives implicit `any`, which `strict` mode rejects.

Literal types and widening

```ts
const a = "left";             // type "left" — a literal type: exactly this string
let b = "left";               // type string — let widens, because it may be reass
let dir: "left" | "right" = "left";     // a union of literals — a lightweight enu
const cfg = { mode: "dark" };            // { mode: string } — widened inside an o
const cfg2 = { mode: "dark" } as const;  // { readonly mode: "dark" } — as const k
const tuple = [1, 2] as const;           // readonly [1, 2]
```

Literal types are how TypeScript models "one of these exact values": `"GET" | "POST"`, `200 | 404`, `true`. `as const` is the tool for configuration objects and tuples that should keep their exact shape. Unions of string literals replace most `enum`s (lesson 2).

`any` — the off switch

`any` is assignable to and from everything and every operation on it is allowed: `const x: any = json; x.foo.bar()` compiles. It is contagious — values derived from `any` are `any` — and it silently removes checking from every expression it touches. Legitimate uses: a migration step from JavaScript, the boundary of an untyped library, a genuinely dynamic value you will not inspect. In each case keep it local and convert to a real type as soon as possible. `strict` mode's `noImplicitAny` refuses *implicit* `any` (un-annotated parameters); explicit `any` is allowed but every occurrence should be a decision.

`unknown` — the type-safe top

`unknown` also accepts every value — but you can do **nothing** with it until you narrow it:

```ts
function handle(data: unknown) {
  data.length;                                  // error: Object is of type 'unkno
  if (typeof data === "string") data.length;    // ok: narrowed to string
  if (Array.isArray(data)) data.map(String);    // ok: any[]
  if (typeof data === "object" && data !== null && "id" in data) …   // narrowed t
}
```

`JSON.parse` returns `any` in the standard library — the first thing to do with it is `as unknown` (or a validator that returns a typed value), so that the compiler forces the checks. `catch (err)` binds `unknown` under `useUnknownInCatchVariables` (part of `strict`), which is why `err.message` needs `err instanceof Error` first. Rule: **`unknown` at boundaries, never `any`.**

`never` — the empty type

`never` has no values. It is the return type of a function that never returns (`throw`, infinite loop), the type of a variable after every possibility has been narrowed away, and the tool for **exhaustiveness**:

```ts
type Shape = { kind: "circle"; r: number } | { kind: "square"; s: number };
function area(s: Shape): number {
  switch (s.kind) {
    case "circle": return Math.PI * s.r ** 2;
    case "square": return s.s ** 2;
    default: { const exhaustive: never = s; throw new Error(`unhandled ${exhaustiv
  }
}
```

Add a third shape and forget a case: `s` in the `default` is no longer `never`, and the assignment fails to compile — the compiler tells you every place that must be updated. `never` in a union disappears (`string | never` is `string`); a function returning `never` lets the compiler know code after it is unreachable.

`void`, `object`, `Function`

`void` is a return type meaning "nothing useful returned" (callers may not use the value); `object` is any non-primitive; `Function` is any callable (too loose — write the signature). Prefer `undefined` returns typed as `void` and specific object shapes over `object`.

Types in JavaScript files

The same checker runs on `.js` with `// @ts-check` at the top (or `checkJs` in tsconfig) using **JSDoc** annotations:

```
// @ts-check
/** @param {number} a @param {number} b @returns {number} */
function add(a, b) { return a + b; }
/** @type {{ name: string; tags: string[] }} */
const user = { name: "Ada", tags: [] };
```

This is the zero-build way to get most of TypeScript's checking in a JavaScript project, and the usual first step of a migration.

Common mistakes

- Trusting an annotation on data that came from outside (`const u = JSON.parse(text) as User` — a lie the compiler cannot catch).
- `any` to silence an error instead of understanding it; `any` spreading through a codebase.
- Annotating everything (`const n: number = 1`) or nothing (implicit `any` parameters).
- Forgetting `as const` on config objects, then fighting `string` where `"dark" | "light"` was meant.
- Missing the exhaustive `never` check on a discriminated union.
- Believing types add runtime safety — they add compile-time checking only.

Interview angle

- *"What does TypeScript add, and what happens at run time?"* Static types checked at compile time, then erased; the emitted JavaScript has no types.
- *"`any` versus `unknown`?"* Both accept anything; `any` disables checking on use, `unknown` requires narrowing before use.
- *"What is `never` for?"* The empty type: functions that do not return, exhausted unions, exhaustiveness checks.
- *"`const a = 'x'` versus `let a = 'x'`?"* `"x"` (literal) versus `string` (widened); `as const` prevents widening in objects and arrays.
- *"How do you type-check plain JavaScript?"* `// @ts-check` with JSDoc annotations, or `checkJs`.

Key takeaways

- Types are compile-time promises, erased at build; the runtime is plain JavaScript — check values at the boundaries.
- Annotate parameters and public returns; let inference do the rest; unions and tuples describe shapes precisely.

`const` gives literal types, `let` widens, `as const` freezes shapes into readonly literals.

`any` disables checking (keep it local and rare); `unknown` forces narrowing (use it at boundaries); `never` marks impossibility and powers exhaustiveness.

`// @ts-check` + JSDoc brings the checker to JavaScript files.

EXERCISE 1

What would TypeScript infer?

Each line is a declaration: `const x = 5`, `let y = "hi"`, `const arr = [1, "a"]`, `const t = [1, 2] as const`, `const o = { "mode": "dark", "n": 1 }`, `let u`. Using the starter's `typeOf(value, { literal, readonly })`, print `<name>: <type>` following TypeScript's rules: `const` keeps a **primitive** literal type (`5`, `"a"`), `let` widens it (`number`, `string`); array elements always widen and join into a union in the order string, number, boolean, null, undefined; object properties widen unless `as const`, which keeps literals, marks properties `readonly` and turns arrays into readonly tuples; a declaration with no initialiser is `any`; an empty array is `never[]`.

Example →

```
x: 5
y: string
arr: (string | number)[]
t: readonly [1, 2]
o: { mode: string; n: number }
o2: { readonly mode: "dark" }
u: any   (declared without an initialiser: implicit any)
```

INPUT 1

```
const x = 5
let y = "hi"
const arr = [1, "a"]
const t = [1, 2] as const
let flag = true
const o = { "mode": "dark", "n": 1 }
const o2 = { "mode": "dark" } as const
let u
const z = null
let empty = []
```

OUTPUT 1

```
x: 5
y: string
arr: (string | number)[]
t: readonly [1, 2]
flag: boolean
o: { mode: string; n: number }
o2: { readonly mode: "dark" }
u: any   (declared without an initialiser:
implicit any)
z: null
empty: never[]
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const ORDER = ["string", "number", "boolean", "null", "undefined"];
function typeOf(value, { literal = false, readonly = false } = {}) {
  if (value === null) return "null";
  if (value === undefined) return "undefined";
  if (typeof value === "string") return literal ? JSON.stringify(value) : "string";
  if (typeof value === "number") return literal ? String(value) : "number";
  if (typeof value === "boolean") return literal ? String(value) : "boolean";
  if (Array.isArray(value)) {
    if (readonly) return `readonly [${value.map((v) => typeOf(v, { literal, readonly })).
    const members = [...new Set(value.map((v) => typeOf(v, { literal: false, readonly }))
    if (members.length === 0) return "never[]";
    return members.length === 1 ? `${members[0]}[]` : `(${members.join(" | ")})[]`;
  }
  const props = Object.entries(value).map(([k, v]) => `${readonly ? "readonly " : ""}${k}
  return props.length ? `{ ${props.join("; ")} }` : "{}";
}
for (const line of lines) {
  const m = /^(const|let)\s+(\w+)\s*(?:=\s*(.*?))?\s*;?$/.exec(line.trim());
  if (!m) { console.log(`cannot parse: ${line.trim()}`); continue; }
  const [, keyword, name, rhsRaw] = m;
  if (rhsRaw === undefined) { console.log(`${name}: any   (declared without an initialise
  const asConst = /\s+as\s+const$/.test(rhsRaw);
  const rhs = rhsRaw.replace(/\s+as\s+const$/, "").replace(/'/g, '"');
  let value;
  try { value = JSON.parse(rhs); } catch { console.log(`${name}: cannot evaluate ${rhs}`)
  const isPrimitive = value === null || typeof value !== "object";
  const literal = asConst || (keyword === "const" && isPrimitive);          // const keep
  console.log(`${name}: ${typeOf(value, { literal, readonly: asConst })}`);
}
```

EXERCISE 2

unknown at the boundary

Each line is JSON — treat the parsed value as `unknown` and narrow it step by step in `handle(data)`: `null` → `null: nothing to do`; a string → `string: <length> chars, upper=<UPPER>`; a number → `number: <2dp>` or `number: not finite`; a boolean → `boolean: yes|no`; an array → `array: <n> items, first=<JSON>`; an object with a `kind` → dispatch `circle` (`Shape circle: area=<2dp>`) and `square` (`Shape square: area=<n>`) with `assertNever` in the default (it throws `unhandled value: <JSON>`); any other object → `object: keys=<comma list or (none)>`. Print `narrowed to <result>`, or `error: invalid JSON` / `error: <message>`.

Example: `"hello"`, `42`, `{"kind":"circle","r":2}`, `{"kind":"hexagon","sides":6}`, `not json` →

```
narrowed to string: 5 chars, upper=HELLO
narrowed to number: 42.00
narrowed to Shape circle: area=12.57
error: unhandled value: {"kind":"hexagon","sides":6}
error: invalid JSON
```

INPUT 1

```
"hello"
42
true
null
[1,2,3]
{"kind":"circle","r":2}
{"kind":"square","s":3}
{"kind":"hexagon","sides":6}
{"id":1,"name":"x"}
not json
```

OUTPUT 1

```
narrowed to string: 5 chars, upper=HELLO
narrowed to number: 42.00
narrowed to boolean: yes
narrowed to null: nothing to do
narrowed to array: 3 items, first=1
narrowed to Shape circle: area=12.57
narrowed to Shape square: area=9
error: unhandled value:
{"kind":"hexagon","sides":6}
narrowed to object: keys=id,name
error: invalid JSON
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
function assertNever(x) { throw new Error(`unhandled value: ${JSON.stringify(x)}`); }
function handle(data /* : unknown */) {
  if (data === null) return "null: nothing to do";
  if (typeof data === "string") return `string: ${data.length} chars, upper=${data.toUppe
  if (typeof data === "number") return Number.isFinite(data) ? `number: ${data.toFixed(2)
  if (typeof data === "boolean") return `boolean: ${data ? "yes" : "no"}`;
  if (Array.isArray(data)) return `array: ${data.length} items, first=${JSON.stringify(da
  if (typeof data === "object" && "kind" in data) {
    switch (data.kind) {
      case "circle": return `Shape circle: area=${(Math.PI * data.r ** 2).toFixed(2)}`;
      case "square": return `Shape square: area=${data.s ** 2}`;
      default: return assertNever(data);              // a kind the types did not anticip
    }
  }
  return `object: keys=${Object.keys(data).join(",") || "(none)"}`;
}
for (const line of lines) {
  try {
    const data = JSON.parse(line);                     // any in the library types — trea
    console.log(`narrowed to ${handle(data)}`);
  } catch (err) {
    console.log(`error: ${err instanceof SyntaxError ? "invalid JSON" : err.message}`);
  }
}
```

QUIZ

1. At run time, a TypeScript type annotation…

A. Is checked on every call

B. Does not exist — types are erased at build; the emitted JavaScript has no trace of them

C. Becomes a runtime assertion

D. Slows the program

2. `const a = "left"` versus `let b = "left"` infer…

A. Both `string`

B. `"left"` (a literal type) and `string` (widened, since `b` may be reassigned)

C. Both `"left"`

D. `any`

3. `any` versus `unknown`:

A. Synonyms

B. Both accept every value; `any` lets you do anything with it unchecked, `unknown` forces you to narrow before use

C. `unknown` accepts fewer values

D. `any` is stricter

4. `never` is the type of…

A. Optional values

B. No value at all: a function that never returns, or a variable after every union member has been narrowed away — the exhaustiveness tool

C. `null`

D. Empty strings

5. To get most of TypeScript's checking in a plain `.js` file you add…

A. A `.ts` extension only

B. `// @ts-check` at the top (or `checkJs`) with JSDoc annotations

C. A `tsconfig.json`

D. Nothing — it is automatic

ANSWER KEY · WHAT TYPESCRIPT ADDS — ERASED TYPES, INFERENCE, AND ANY VERSUS UNKNOWN VERSUS NEVER

1. B Which is why values from JSON or users must be validated at the boundary.

2. B `as const` keeps literals inside objects and arrays too.

3. B `unknown` at boundaries; `any` only as a local, deliberate escape hatch.

4. B Assigning the leftover to `never` in a `default` fails to compile when a case is missing.

5. B The usual first step of a migration.

LESSON 2 · 13 MIN

Object types, interfaces, structural typing and discriminated unions

Most of a TypeScript program's types describe **objects**: what properties they have, which are optional or read-only, what shape each is. Two features decide how those types behave and both surprise newcomers from Java or C#: types are **structural** (a value matches a type if it has the right shape, whatever it was declared as), and the natural way to model "one of several kinds of thing" is not a class hierarchy but a **discriminated union** — a `kind` field the compiler can switch on. This lesson covers object types and interfaces, optional/readonly/index signatures, `type` versus `interface`, structural assignability and excess-property checks, intersections, discriminated unions, `enum` versus literal unions, and `satisfies`.

Object types

```
interface User {
  id: string;
  name: string;
  email?: string;                // optional: string | undefined
  readonly createdAt: Date;      // cannot be reassigned after construction
  tags: readonly string[];       // a readonly array: no push/pop/assignment by i
  [extra: string]: unknown;      // index signature: any other string key holds a
}
type Point = { x: number; y: number };          // type alias — same for object sh
```

Optional (`?`) means the property may be absent or `undefined`; under `exactOptionalPropertyTypes` the two are distinguished. `readonly` is compile-time only (no `Object.freeze`). An index signature says "any key of this form maps to this type" — for dictionaries, though `Record<string, T>` or a `Map` is usually clearer. Method syntax `greet(): string` and property syntax `greet: () ⇒ string` both work; the latter is stricter about parameter variance.

`interface` versus `type`

Both name an object shape. `interface` can be **extended** (`interface Admin extends User { role: "admin" }`), **merged** across declarations (two `interface Window` blocks combine — how libraries augment globals), and gives clearer error messages for object shapes. `type` can name **anything** — unions, tuples, primitives, function types, mapped and conditional types — and uses `&` for intersection. Convention in most codebases: `interface` for object shapes that may be extended or implemented by classes, `type` for unions, aliases and computed types. Either is fine; be consistent.

Structural typing

```
interface Named { name: string }
const dog = { name: "Rex", legs: 4 };
const n: Named = dog;                    // ok — dog HAS a name; extra properties
function greet(x: Named) {}
greet({ name: "Ada", age: 36 });         // error: excess property 'age' — a fresh
greet(dog);                              // ok — the same object through a variabl
```

A type is a set of required properties; any value with those (and any others) is assignable. Names do not matter — two interfaces with identical members are interchangeable. This is why a plain

object can satisfy an interface without `implements`, why library types compose without inheritance, and why "duck typing" is checked statically. The one place TypeScript is *nominal-ish* is **excess property checking**: passing an object **literal** directly to a typed slot flags unknown properties (usually a typo), while the same object through a variable is accepted. Classes are structural too — `class A { x = 1 }` and `class B { x = 1 }` are assignable to each other — unless they have private members, which are compared by declaration.

Intersections and composition

```
type Timestamped = { createdAt: Date; updatedAt: Date };
type Post = { title: string; body: string } & Timestamped;        // has all five
type Handler = ((e: Event) ⇒ void) & { displayName: string };     // a callable wi
```

`&` combines shapes; conflicting property types intersect too (`string & number` is `never` — a common source of "why is this never"). Intersections compose behaviour-less data shapes the way mixins compose classes.

Discriminated unions

```
type Shape =
  | { kind: "circle"; radius: number }
  | { kind: "rect"; width: number; height: number }
  | { kind: "square"; size: number };

function area(s: Shape): number {
  switch (s.kind) {                          // the discriminant: a literal-typed
    case "circle": return Math.PI * s.radius ** 2;   // s is narrowed to the circl
    case "rect": return s.width * s.height;
    case "square": return s.size ** 2;
  }
}
```

Each member carries a literal `kind` (or `type`, `status`, `tag`); a `switch` / `if` on it **narrows** the union so the member's own properties become available. This is the idiomatic replacement for class hierarchies with `instanceof`, for "objects with a type string and a bag of optional fields", and for state modelling (`{ status: "loading" } | { status: "error"; error: Error } | { status: "ready"; data: T }` — impossible states become unrepresentable). Add the `never` default (previous lesson) for exhaustiveness. Result-style types (module 11) are discriminated unions on `ok: true | false`.

Unions of primitives and literal unions

`type Id = string | number` — narrow with `typeof`. `type Method = "GET" | "POST" | "PUT" | "DELETE"` — a closed set of strings the compiler completes and checks; combine with `as const` arrays to derive the type from the data: `const METHODS = ["GET", "POST"] as const; type Method = (typeof METHODS)[number];`.

`enum` — and why unions usually win

```
enum Color { Red, Green }           // numeric: Color.Red ═══ 0, and any number is
enum Level { Low = "low", High = "high" }   // string enum: a real runtime object,
```

Enums generate runtime code (an object), are nominal (a `"low"` string is **not** assignable to `Level`), and numeric enums accept arbitrary numbers. String-literal unions are erased, structural, need no import at call sites, and give the same autocompletion. Most modern style guides prefer unions (or `as const` objects) and reserve enums for interop with existing enum-shaped data. `const enum` inlines values but breaks under isolated-module builds — avoid.

`satisfies`

```
const palette = { primary: "#2563EB", muted: "#64748B" } satisfies Record<string,
palette.primary.toUpperCase();     // still knows primary is a string literal-type
```

`satisfies` (4.9) checks a value against a type **without** widening the value to that type — you get validation and keep the precise inferred type. Before it, you chose between `const p: Record<string, string> = …` (loses the exact keys) and no check at all.

Common mistakes

- Expecting a class name to matter for assignability (it does not — shape does).
- Fighting excess-property errors by widening the type instead of fixing the typo.
- Optional-field bags (`{ radius?: number; width?: number; … }`) instead of a discriminated union.
- `string & number` intersections that collapse to `never`.
- Numeric enums; `const enum` in bundled projects.
- Annotating a config object with a wide type and losing literal keys (use `satisfies` or `as const`).

Interview angle

- *"Is TypeScript nominally or structurally typed?"* Structurally — shape decides assignability; excess-property checks on fresh literals and private class members are the exceptions.
- *"`interface` versus `type`?"* Interfaces extend and merge and suit object shapes; types alias anything including unions; pick a convention.
- *"What is a discriminated union?"* A union whose members share a literal-typed tag; switching on it narrows to the member — the idiomatic alternative to class hierarchies and optional bags.
- *"Why prefer string-literal unions to enums?"* Erased, structural, no imports, same autocompletion; enums are nominal runtime objects and numeric ones are unsafe.
- *"What does `satisfies` do?"* Validates a value against a type while keeping the value's inferred (narrower) type.

Key takeaways

- Object types: `?` optional, `readonly`, index signatures; `interface` extends/merges, `type` aliases anything; be consistent.
- Structural typing: shape is everything; fresh literals get excess-property checks; private members make classes nominal.
- `&` intersects shapes (conflicts become `never`); discriminated unions with a literal `kind` model variants and states — switch to narrow, `never` to exhaust.
- String-literal unions (with `as const` sources) over enums.
- `satisfies` validates without widening; `as const` keeps literals.

EXERCISE 1

Structural assignability, with the excess-property rule

Line 1 is a type as JSON — property → type, where a trailing `?` marks optional and types are `string`, `number`, `boolean` or `T[]`. Each following line is `fresh <json>` (an object literal passed directly) or `var <json>` (the same object via a variable). Report, in property order: `Property 'k' is missing` for absent required properties; `Type '<actual>' is not assignable to type '<expected>' at 'k'` for mismatches (an array's actual type is the union of its element types plus `[]`); and — for `fresh` only — `Object literal may only specify known properties, and 'k' does not exist`. Print `<mode> <JSON>: ok` or `… : not assignable - <problems joined by ; >`.

Example with `{"name":"string","age":"number","email?":"string","tags":"string[]"}`:

```
fresh {"name":"Ada","age":36,"tags":["a"],"zip":"123"}: not assignable - Object literal m
var {"name":"Ada","age":36,"tags":["a"],"zip":"123"}: ok
fresh {"name":"Ada","age":"36","tags":[1]}: not assignable - Type 'string' is not assigna
```

The same object is rejected as a fresh literal and accepted through a variable — structural typing with one nominal-ish exception.

INPUT 1

```
{"name":"string","age":"number","email?":"string","tags":"string[]"}
fresh {"name":"Ada","age":36,"tags":["a"]}
fresh {"name":"Ada","age":36,"tags":["a"],"zip":"123"}
var {"name":"Ada","age":36,"tags":["a"],"zip":"123"}
fresh {"name":"Ada","tags":[]}
fresh {"name":"Ada","age":"36","tags":[1]}
var {"name":"Bo","age":1,"email":"b@x","tags":[]}
```

OUTPUT 1

```
fresh {"name":"Ada","age":36,"tags":
["a"]}: ok
fresh {"name":"Ada","age":36,"tags":
["a"],"zip":"123"}: not assignable -
Object literal may only specify known
properties, and 'zip' does not exist
var {"name":"Ada","age":36,"tags":
["a"],"zip":"123"}: ok
fresh {"name":"Ada","tags":[]}: not
assignable - Property 'age' is missing
fresh {"name":"Ada","age":"36","tags":
[1]}: not assignable - Type 'string' is
not assignable to type 'number' at 'age'
Type 'number[]' is not assignable to typ
'string[]' at 'tags'
var
{"name":"Bo","age":1,"email":"b@x","tags
[]}: ok
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const schema = JSON.parse(lines[0]);      // { "name": "string", "email?": "string", "tag
function matchesType(value, type) {
  if (type.endsWith("[]")) return Array.isArray(value) && value.every((v) => matchesType(
  if (type === "null") return value === null;
  return typeof value === type;
}
function check(value, schema, { fresh }) {
  const problems = [];
  const known = new Set();
  for (const [rawKey, type] of Object.entries(schema)) {
    const optional = rawKey.endsWith("?");
    const key = optional ? rawKey.slice(0, -1) : rawKey;
    known.add(key);
    if (!(key in value)) { if (!optional) problems.push(`Property '${key}' is missing`);
    if (!matchesType(value[key], type)) problems.push(`Type '${typeOfValue(value[key])}'
  }
  if (fresh) for (const key of Object.keys(value)) if (!known.has(key)) problems.push(`Ob
  return problems;
}
const typeOfValue = (v) => (v === null ? "null" : Array.isArray(v) ? `${[...new Set(v.map
for (const line of lines.slice(1)) {
  const [mode, ...rest] = line.trim().split(/\s+/);
  const value = JSON.parse(rest.join(" "));
  const problems = check(value, schema, { fresh: mode === "fresh" });
  console.log(`${mode} ${JSON.stringify(value)}: ${problems.length ? `not assignable - ${
}
```

EXERCISE 2

Dispatch on a discriminant, exhaustively

Write `area(shape)` for `{ kind: "circle", r } | { kind: "rect", w, h } | { kind: "square", s }` and `render(state)` for `{ status: "loading" } | { status: "error", error } | { status: "ready", data }` (→ `spinner`, `error banner: <error>`, `list of <data.length>`), each a `switch` on the discriminant with `assertNever` in the `default` (it throws `unhandled variant: <JSON>`). For each JSON line: objects with `kind` print `<kind>: area=<2dp>`, objects with `status` print `<status>: <render>`, anything else `no discriminant: cannot dispatch`; thrown errors print `error: <message>`.

Example: `{"kind":"circle","r":1}`, `{"kind":"triangle","b":1,"h":2}`, `{"status":"error","error":"timeout"}`, `{"status":"paused"}` →

```
circle: area=3.14
error: unhandled variant: {"kind":"triangle","b":1,"h":2}
error: error banner: timeout
error: unhandled variant: {"status":"paused"}
```

INPUT 1

```
{"kind":"circle","r":1}
{"kind":"rect","w":2,"h":3}
{"kind":"square","s":4}
{"kind":"triangle","b":1,"h":2}
{"status":"loading"}
{"status":"error","error":"timeout"}
{"status":"ready","data":[1,2,3]}
{"status":"paused"}
{"x":1}
```

OUTPUT 1

```
circle: area=3.14
rect: area=6.00
square: area=16.00
error: unhandled variant:
{"kind":"triangle","b":1,"h":2}
loading: spinner
error: error banner: timeout
ready: list of 3
error: unhandled variant:
{"status":"paused"}
no discriminant: cannot dispatch
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
function assertNever(x) { throw new Error(`unhandled variant: ${JSON.stringify(x)}`); }
function area(shape) {
  switch (shape.kind) {
    case "circle": return Math.PI * shape.r ** 2;      // shape is the circle member here
    case "rect": return shape.w * shape.h;
    case "square": return shape.s ** 2;
    default: return assertNever(shape);
  }
}
function render(state) {
  switch (state.status) {
    case "loading": return "spinner";
    case "error": return `error banner: ${state.error}`;
    case "ready": return `list of ${state.data.length}`;
    default: return assertNever(state);
  }
}
for (const line of lines) {
  try {
    const value = JSON.parse(line);
    if ("kind" in value) console.log(`${value.kind}: area=${area(value).toFixed(2)}`);
    else if ("status" in value) console.log(`${value.status}: ${render(value)}`);
    else console.log("no discriminant: cannot dispatch");
  } catch (err) {
    console.log(`error: ${err.message}`);
  }
}
```

QUIZ

1. `interface Named { name: string }` and `const dog = { name: "Rex", legs: 4 }; const n: Named = dog;` …

   A. Fails — `dog` is not declared as `Named`

   B. Compiles — structural typing: `dog` has a `name`; extra properties are fine through a variable

   C. Fails because of `legs`

   D. Needs `implements`

2. A discriminated union is…

   A. A union of primitives

   B. A union of object types sharing a literal-typed tag (`kind`, `status`); switching on the tag narrows to the member

   C. An enum

   D. An intersection

3. `interface` versus `type`:

   A. Identical in every way

   B. Interfaces extend and merge and suit object shapes; type aliases name anything (unions, tuples, mapped types)

   C. `type` is deprecated

   D. Only interfaces work with classes

4. Why do most style guides prefer `type Level = "low" | "high"` to an `enum`?

A. Enums are slower

B. The union is erased, structural and needs no import; enums are nominal runtime objects and numeric enums accept any number

C. Enums cannot be strings

D. Unions autocomplete better

5. `x satisfies T` differs from `x as T` in that it…

A. Widens `x` to `T`

B. Checks `x` against `T` while keeping `x`'s narrower inferred type — validation without widening

C. Is a runtime check

D. Only works on primitives

ANSWER KEY · OBJECT TYPES, INTERFACES, STRUCTURAL TYPING AND DISCRIMINATED UNIONS

1. B Passing the object **literal** directly would trigger the excess-property check.

2. B The idiomatic replacement for class hierarchies and bags of optional fields.

3. B Pick a convention and stay consistent.

4. B Derive the union from an `as const` array when the values are also data.

5. B `const palette = {…} satisfies Record<string, string>` keeps the exact keys.

LESSON 3 · 13 MIN

Function types, generics, keyof and the utility types

Generics are how TypeScript types code that works for many types without giving up precision: `first(xs: number[]): number` and `first(xs: string[]): string` become one `first<T>(xs: T[]): T`, and the compiler fills in `T` at each call. Combined with `keyof`, indexed access and constraints, generics express the relationships real code has — "this key must exist on that object", "the return type is the property's type" — and the standard library's **utility types** (`Partial`, `Pick`, `Record`, `ReturnType` …) are generics you will use daily. This lesson covers function typing, generic functions and constraints, `keyof`/indexed access, the utility types with what each is made of, and a first look at mapped and conditional types so their error messages make sense.

Typing functions

```
function add(a: number, b: number): number { return a + b; }
const mul = (a: number, b: number): number => a * b;
type BinaryOp = (a: number, b: number) => number;            // a function type
function apply(op: BinaryOp, a: number, b: number) { return op(a, b); }

function greet(name: string, greeting = "Hello", ...rest: string[]): string { … }
function log(msg: string, level?: "info" | "warn") {}
```

Return types are inferred; annotate them on exported functions so a change inside does not silently change the public contract. Parameters are checked **contravariantly-ish** (a callback declared to take `Animal` may be given one taking `Dog` only under `strictFunctionTypes` rules for method syntax) — in practice: callbacks may accept *fewer* parameters than provided (`arr.map((x) => …)` ignores index) but not more.

Overloads describe a function whose return type depends on argument types:

```
function parse(x: string): number;
function parse(x: number): string;
function parse(x: string | number) { return typeof x === "string" ? Number(x) : St
```

Prefer a union or a generic when it expresses the same thing; overloads for genuinely different signatures.

Generic functions

```
function first<T>(xs: T[]): T | undefined { return xs[0]; }
first([1, 2]);            // T inferred as number → number | undefined
first(["a"]);             // string | undefined
first<number>([]);        // explicit

function map<T, U>(xs: T[], f: (x: T) => U): U[] { return xs.map(f); }
const lens = map(["a", "bb"], (s) => s.length);   // T = string, U = number → numb
```

A type parameter stands for "some type, decided by the caller"; inference picks it from the arguments. Name them `T`, `U`, `K`, `V` for simple cases and descriptively (`TItem`, `TKey`) when several interact. A function that does not use `T` in its parameters cannot infer it — that is a sign the generic is misplaced.

Constraints, `keyof` and indexed access

```
function longest<T extends { length: number }>(a: T, b: T): T { return a.length ⩾
longest("abc", "de");  longest([1, 2], [3]);   longest(1, 2);   // error: number h

function pluck<T, K extends keyof T>(items: T[], key: K): T[K][] { return items.ma
pluck(users, "name");     // string[] — K is "name", T[K] is string
pluck(users, "zip");      // error: Argument of type '"zip"' is not assignable to
```

`extends` **constrains** a type parameter (the value must at least have this shape). `keyof T` is the union of `T`'s property names as literal types (`"id" | "name" | "email"`). `T[K]` (indexed access) is the type of property `K` on `T`. Together they type "give me the property named `key`" so that a misspelled key is a compile error and the return type follows the property — the pattern behind every typed `get`, `groupBy(key)`, `sortBy(key)`, form-field binding and ORM column reference.

Generic types and classes

```
type Result<T, E = Error> = { ok: true; value: T } | { ok: false; error: E };    /
class Stack<T> {
  #items: T[] = [];
  push(x: T) { this.#items.push(x); }
  pop(): T | undefined { return this.#items.pop(); }
}
const s = new Stack<number>(); s.push("x");     // error
interface Repository<T extends { id: string }> { get(id: string): Promise<T | unde
```

`Array<T>`, `Promise<T>`, `Map<K, V>`, `Set<T>`, `Record<K, V>` are generic types you already use; `Promise<User[]>` reads as "a promise of an array of users".

The utility types

All built from mapped and conditional types (below); know what each does and roughly how:

Utility	Meaning	Made of
`Partial<T>`	every property optional	`{ [K in keyof T]?: T[K] }`
`Required<T>`	every property required	`{ [K in keyof T]-?: T[K] }`
`Readonly<T>`	every property readonly	`{ readonly [K in keyof T]: T[K] }`
`Pick<T, K>`	only the keys `K`	`{ [P in K]: T[P] }`
`Omit<T, K>`	all but the keys `K`	`Pick<T, Exclude<keyof T, K>>`
`Record<K, V>`	an object with keys `K` and values `V`	`{ [P in K]: V }`
`Exclude<T, U>` / `Extract<T, U>`	remove / keep union members assignable to `U`	conditional over the union
`NonNullable<T>`	drop `null` and `undefined`	`T & {}`
`ReturnType<F>` / `Parameters<F>`	a function's return / parameter tuple	`infer`
`Awaited<T>`	what `await` yields	recursive conditional

`Partial<User>` for update payloads, `Pick<User, "id" | "name">` for a list view, `Omit<User, "passwordHash">` for a public shape, `Record<Method, Handler>` for a dispatch table, `ReturnType<typeof makeStore>` to name a type you never wrote.

Mapped and conditional types, briefly

```
type Nullable<T> = { [K in keyof T]: T[K] | null };            // mapped: transfor
type Getters<T> = { [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K] }
type ElementOf<A> = A extends (infer E)[] ? E : never;          // conditional wit
type IsString<T> = T extends string ? "yes" : "no";             // distributes ove
```

A **mapped type** iterates `keyof` and builds a new object type; a **conditional type** chooses between types with `extends ? :` and can `infer` parts of a type. They are the meta-programming layer of the type system — you read them in library types and error messages long before you write them; when you do write one, keep it small and name it.

Common mistakes

- `any` in a function signature because "it takes anything" — that is `unknown` or a generic.
- A generic whose `T` appears only in the return type (uninferrable).
- `keyof` without a constraint (`function get<T>(o: T, k: string)` — `o[k]` is an error; use `K extends keyof T`).
- `Omit` on a union (it operates on the union's *common* keys — use a distributive helper).
- Reaching for overloads where a union parameter suffices; or for conditional types where a plain union would do.
- Under-typing callbacks (`(x: any) => void`) in generic helpers.

Interview angle

- *"What is a generic?"* A type parameter decided per use, letting one function or type work for many types while staying precise.
- *"Explain `K extends keyof T` and `T[K]`."* Constrain the key to the object's property names; index the object type by it to get that property's type.
- *"`Partial` versus `Pick` versus `Omit`?"* All optional; only these keys; all but these keys.
- *"How is `Omit` implemented?"* `Pick<T, Exclude<keyof T, K>>`.
- *"What is a mapped type?"* A type built by iterating `keyof` some type and transforming each property.

Key takeaways

- Type parameters and returns explicitly on public functions; function types describe callbacks; overloads only for genuinely different signatures.
- Generics infer `T` from arguments; constrain with `extends`; `keyof T` + `T[K]` type property access safely.
- Generic types/classes: `Result<T, E>`, `Stack<T>`, `Repository<T extends { id }>`.
- Utility types: `Partial`, `Required`, `Readonly`, `Pick`, `Omit`, `Record`, `Exclude` / `Extract`, `NonNullable`, `ReturnType` / `Parameters`, `Awaited`.
- Mapped types transform properties; conditional types choose with `extends ? :` and `infer` — read them, write them sparingly.

EXERCISE 1

Utility types as operations on a schema

Line 1 is a schema — property → type text, with a trailing `?` for optional and a leading `readonly` for readonly. Implement the utilities as functions on schemas and print the resulting TypeScript text `{ k: t; … }`: `Partial` (add `?`), `Required` (remove it), `Readonly` (prefix `readonly `), `Pick k k…` (only those keys; an unknown key throws `Type '"k"' does not satisfy the constraint 'keyof T'`), `Omit k k…`, `Record k v` (→ `{ [key in k]: v }`), `keyof` (→ `"a" | "b"`, or `never`). Steps chain with `|`. Print `<command> ⇒ <type>` or `<command> ⇒ error: <message>`.

Example with `{"id":"string","name":"string","email?":"string","age":"number"}`:

```
Partial ⇒ { id?: string; name?: string; email?: string; age?: number }
Pick id name ⇒ { id: string; name: string }
Omit age | Partial ⇒ { id?: string; name?: string; email?: string }
Pick id email | Readonly | keyof ⇒ "id" | "email"
Pick zip ⇒ error: Type '"zip"' does not satisfy the constraint 'keyof T'
```

INPUT 1

```
{"id":"string","name":"string","email?":"string","age":"number"}
Partial
Required
Readonly
Pick id name
Omit age
Record status number
keyof
Omit age | Partial
Pick id email | Readonly | keyof
Pick zip
Frobnicate
```

OUTPUT 1

```
Partial ⇒ { id?: string; name?: string; email?: string; age?: number }
Required ⇒ { id: string; name: string; email: string; age: number }
Readonly ⇒ { readonly id: string; readonly name: string; readonly email?: string; readonly age: number }
Pick id name ⇒ { id: string; name: string }
Omit age ⇒ { id: string; name: string; email?: string }
Record status number ⇒ { [key in status]: number }
keyof ⇒ "id" | "name" | "email" | "age"
Omit age | Partial ⇒ { id?: string; name?: string; email?: string }
Pick id email | Readonly | keyof ⇒ "id" | "email"
Pick zip ⇒ error: Type '"zip"' does not satisfy the constraint 'keyof T'
Frobnicate ⇒ error: unknown utility 'Frobnicate'
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
// A schema is { key: type } with a trailing ? for optional and a leading `readonly ` for
const schema = JSON.parse(lines[0]);
const print = (s) => `{ ${Object.entries(s).map(([k, t]) => `${k}: ${t}`).join("; ")} }`;
const base = (k) => k.replace(/^readonly /, "").replace(/\?$/, "");
const utilities = {
  Partial: (s) => Object.fromEntries(Object.entries(s).map(([k, t]) => [k.endsWith("?") ?
  Required: (s) => Object.fromEntries(Object.entries(s).map(([k, t]) => [k.replace(/\?$/,
  Readonly: (s) => Object.fromEntries(Object.entries(s).map(([k, t]) => [k.startsWith("re
  Pick: (s, keys) => {
    for (const key of keys) if (!Object.keys(s).some((k) => base(k) === key)) throw new T
    return Object.fromEntries(Object.entries(s).filter(([k]) => keys.includes(base(k))));
  },
  Omit: (s, keys) => Object.fromEntries(Object.entries(s).filter(([k]) => !keys.includes(
  Record: (_s, [k, v]) => ({ [`[key in ${k}]`]: v }),
  keyof: (s) => Object.keys(s).map((k) => `"${base(k)}"`).join(" | ") || "never",
};
for (const line of lines.slice(1)) {
  try {
    let current = schema;
    for (const step of line.split("|").map((s) => s.trim()).filter(Boolean)) {
      const [name, ...args] = step.split(/\s+/);
      if (!utilities[name]) throw new TypeError(`unknown utility '${name}'`);
      current = utilities[name](current, args);
    }
    console.log(`${line.trim()} => ${typeof current === "string" ? current : print(curren
  } catch (err) {
    console.log(`${line.trim()} => error: ${err.message}`);
  }
}
```

EXERCISE 2

Generic constraints, enforced at run time

Model three generic signatures as runtime-checked functions: `first<T>(xs: T[]): T | undefined`; `pluck<T, K extends keyof T>(items: T[], key: K): T[K][]` — every item must have the key, else throw `Argument of type '"<key>"' is not assignable to parameter of type 'keyof T'`; `longest<T extends { length: number }>(a: T, b: T): T` — both need a numeric `length`, else throw `Argument of type '<type>' is not assignable to parameter of type '{ length: number; }'`. Line 1 is the items array; commands `first` (prints the value and `: T` or `: undefined (T | undefined)`), `pluck <key>` (prints the array and its element type as `<union>[]`, `never[]` when empty), `longest <json> <json>`. Errors print the TypeScript-style message.

Example with two user records:

```
first → {"id":"a","name":"Ada","age":36} : T
pluck name → ["Ada","Bob"] : string[]
pluck zip → Argument of type '"zip"' is not assignable to parameter of type 'keyof T'
longest "abc" "de" → "abc"
longest 1 2 → Argument of type 'number' is not assignable to parameter of type '{ length
```

INPUT 1

```
[{"id":"a","name":"Ada","age":36},
{"id":"b","name":"Bob","age":17}]
first
pluck name
pluck age
pluck zip
longest "abc" "de"
longest [1,2,3] [4]
longest 1 2
```

OUTPUT 1

```
first → {"id":"a","name":"Ada","age":36}
: T
pluck name → ["Ada","Bob"] : string[]
pluck age → [36,17] : number[]
pluck zip → Argument of type '"zip"' is
not assignable to parameter of type 'keyof
T'
longest "abc" "de" → "abc"
longest [1,2,3] [4] → [1,2,3]
longest 1 2 → Argument of type 'number'
is not assignable to parameter of type '{
length: number; }'
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
// Runtime stand-ins for generic signatures:  first<T>(xs: T[]): T | undefined   pluck<T,
const tsType = (v) => (v === null ? "null" : Array.isArray(v) ? "array" : typeof v);
function first(xs) { return xs[0]; }
function pluck(items, key) {
  if (!items.every((it) => it !== null && typeof it === "object" && key in it)) throw new
  return items.map((it) => it[key]);      // T[K][]: the element type follows the propert
}
function longest(a, b) {
  for (const v of [a, b]) if (v === null || typeof v.length !== "number") throw new TypeE
  return a.length >= b.length ? a : b;
}
const items = JSON.parse(lines[0]);
for (const line of lines.slice(1)) {
  const [cmd, ...args] = line.trim().split(/\s+/);
  try {
    if (cmd === "first") { const v = first(items); console.log(`first → ${JSON.stringify
    else if (cmd === "pluck") { const out = pluck(items, args[0]); console.log(`pluck ${a
    else if (cmd === "longest") { const [a, b] = args.map((s) => JSON.parse(s)); console.
  } catch (err) {
    console.log(`${line.trim()} → ${err.message}`);
  }
}
```

QUIZ

1. `function pluck<T, K extends keyof T>(items: T[], key: K): T[K][]` — `K extends keyof T` means…

A. `K` is a string

B. `K` must be one of `T`'s property names, so a misspelled key is a compile error and the return type follows the property

C. `K` is optional

D. `K` is a subclass

2. `Omit<T, K>` is implemented as…

A. A compiler intrinsic

B. `Pick<T, Exclude<keyof T, K>>`

C. `Partial<T>` minus `K`

D. `Record<K, never>`

3. A generic whose type parameter appears only in the return type…

A. Is the best design

B. Cannot be inferred from a call — the caller must always pass it explicitly; usually a misplaced generic

C. Is faster

D. Defaults to `any`

4. `Partial<User>` is typically used for…

A. Deleting users

B. Update payloads where every field is optional

C. Making users readonly

D. Runtime validation

5. A mapped type…

A. Is a `Map<K, V>`

B. Builds a new object type by iterating `keyof` another type and transforming each property (`{ [K in keyof T]?: T[K] }`)

C. Maps arrays

D. Is a runtime construct

ANSWER KEY · FUNCTION TYPES, GENERICS, KEYOF AND THE UTILITY TYPES

1. B `T[K]` is indexed access: the type of that property.

2. B Most utility types are small mapped or conditional types.

3. B Inference comes from the arguments.

4. B `Pick` for a list view, `Omit<User, "passwordHash">` for a public shape, `Record<Method, Handler>` for a dispatch table.

5. B Conditional types choose with `extends ? :`; read them in library types before writing them.

LESSON 4 · 12 MIN

Narrowing, type guards, assertions — making the compiler trust a value

A union type says "one of these"; before you can use the value as one particular member you must **narrow** it, and TypeScript understands the same checks you would write in plain JavaScript — `typeof`, `instanceof`, `in`, equality, truthiness, a `switch` on a discriminant. Narrowing is control-flow analysis: inside the `if`, the variable has a smaller type; in the `else`, the complement. When the built-in checks cannot express what you know, you write a **type guard** (`x is User`) or an **assertion function**, and when you know better than the compiler you use `as` — carefully, because `as` is a promise the compiler will not verify. This lesson covers every narrowing form, exhaustiveness, guards and assertions, the two escape hatches and their rules, and how runtime validation at the boundary connects to all of it.

Narrowing by control flow

```
function describe(x: string | number | null | Date | string[]) {
  if (x === null) return "null";                      // equality narrowing: x is
  if (typeof x === "string") return x.toUpperCase();  // typeof: "string" | "numb
  if (typeof x === "number") return x.toFixed(2);
  if (x instanceof Date) return x.toISOString();      // instanceof: classes and
  if (Array.isArray(x)) return x.join(",");           // built-in guards: Array.i
  return x;                                           // never — everything has b
}
```

After each `return`, the remaining type shrinks. `typeof x === "object"` leaves `null` in (it is `"object"` too) — check `x !== null` first or together. **Truthiness** narrowing (`if (x)`) removes `null` / `undefined` but also `0`, `""` and `false` — fine for objects, a bug for numbers and strings that may legitimately be falsy. Narrowing applies to `const`s and to properties along a path (`obj.a.b`) until something could have changed them (a function call, an `await`, an assignment) — the compiler then resets to the declared type, which is why extracting to a local `const` sometimes "fixes" a narrowing error.

`in`, discriminants and `switch`

```
type Cat = { meow(): void }; type Dog = { bark(): void };
function speak(p: Cat | Dog) { if ("meow" in p) p.meow(); else p.bark(); }   // `i

type Event = { type: "click"; x: number } | { type: "key"; key: string } | { type:
function handle(e: Event) {
  switch (e.type) {
    case "click": return e.x;
    case "key": return e.key;
    case "scroll": return e.dy;
    default: return assertNever(e);      // compile error the day a fourth event a
  }
}
function assertNever(x: never): never { throw new Error(`unexpected: ${JSON.string
```

The discriminant switch is the workhorse of typed application code; `assertNever` turns "I forgot a case" into a build failure and, at run time, into a loud error for data the types did not anticipate.

User-defined type guards

```ts
interface User { id: string; name: string; email?: string }
function isUser(x: unknown): x is User {
  return typeof x === "object" && x !== null
    && typeof (x as any).id === "string"
    && typeof (x as any).name === "string"
    && ((x as any).email === undefined || typeof (x as any).email === "string");
}
const data: unknown = JSON.parse(text);
if (isUser(data)) data.name;      // narrowed to User
users.filter(isUser);             // filter with a guard returns User[] — plain (u
```

A function returning `x is T` tells the compiler "when I return true, treat `x` as `T`". Its body is ordinary JavaScript — **you** are responsible for it being correct; a guard that checks less than it claims is an `as` in disguise. Guards compose (`isArrayOf(isUser)`), work with `filter`, and are the honest way to turn `unknown` into a type. For anything beyond a few fields use a schema library (zod, valibot) whose `parse` returns the typed value and whose type you `infer` — one definition, runtime check and static type together.

Assertion functions

```ts
function assertIsUser(x: unknown): asserts x is User { if (!isUser(x)) throw new T
function assert(cond: unknown, msg?: string): asserts cond { if (!cond) throw new

assertIsUser(data);   data.name;          // narrowed from here on — no if block
assert(user, "user required"); user.id;   // removes undefined
```

`asserts x is T` narrows for the rest of the scope when the function returns; `asserts cond` narrows on any boolean expression (`assert(x !== null)`). Node's `assert` module is typed this way. Use assertions for invariants (module 7's distinction): a failure is a bug, and the throw is the right outcome.

`as` and `!` — the escape hatches

```ts
const input = document.getElementById("q") as HTMLInputElement;   // you know more
const el = document.querySelector(".x")!;                          // non-null ass
const n = value as unknown as number;                              // double asser
```

`as` narrows or widens between **compatible** types without any check; `!` removes `null | undefined`. Both compile to nothing. Acceptable when the knowledge is real and local (DOM element types, a value you just validated, a literal you want narrowed); a bug source when used to silence an error about data from outside. Rules: prefer a guard or a schema when the value came from I/O; prefer narrowing (`if`) to `!`; never `as any`; treat every `as` in a diff as a review question.

Narrowing `unknown` at the boundary

The pattern that makes the whole type system honest:

1. Data enters as `unknown` (`JSON.parse(text) as unknown`, `req.body as unknown`, `catch (err: unknown)`).
2. A guard or schema **validates** it and returns a typed value (or throws/returns an error).
3. Everything inside works with the typed value; no `as`, no `any`.

For errors: `if (err instanceof Error) err.message`, or a guard for your `AppError` (module 7). For `catch`, `strict` binds `err` as `unknown` for exactly this reason.

Common mistakes

`typeof x === "object"` forgetting `null`; truthiness narrowing on numbers/strings.

Narrowing lost across a function call or `await` (extract to a `const`).

A type guard that returns true without checking everything it claims.

`as` to convert I/O data (`JSON.parse(t) as User`) — no check happened.

`!` sprinkled to silence strict null errors instead of handling the null case.

Missing `assertNever` in exhaustive switches; `default: return undefined` hiding new cases.

Interview angle

"What is narrowing?" Control-flow analysis that shrinks a union inside a branch based on `typeof`, `instanceof`, `in`, equality, truthiness or a discriminant check.

"What is a type guard?" A function returning `x is T` whose true result narrows `x`; you write the runtime check.

"When is `as` acceptable?" When you know something the compiler cannot and the knowledge is local and safe (DOM types, just-validated data); never for I/O data.

"How do you handle `unknown` from JSON?" Validate with a guard or schema at the boundary and work with the typed result.

"What does `asserts x is T` do?" Narrows `x` for the rest of the scope after the function returns (it throws otherwise).

Key takeaways

`typeof`, `instanceof`, `in`, `===`, truthiness and discriminant `switch` narrow by control flow; narrowing resets across calls and `await`.

`assertNever` in the `default` makes unions exhaustive at compile time and loud at run time.

Type guards (`x is T`) and assertion functions (`asserts x is T`) are the honest bridges from `unknown`; their bodies must check what they claim.

`as` and `!` are unchecked promises — local, rare, never on I/O data; `as any` never.

Boundary pattern: `unknown` in → validate (guard/schema) → typed everywhere else.

EXERCISE 1

A narrowing engine over a union

Line 1 lists the members of a union: `string number boolean null undefined Date string[] {kind:"a"} {kind:"b"}`. Each following line is a check: `typeof <t>`, `instanceof Date`, `=== null`, `truthy`, `Array.isArray`, `in kind`, `kind === <v>`. For each, compute the members remaining in the true branch and in the false branch and print `<check> → true: <members or never> | false: <members or never>`. Rules: `typeof null` is `"object"`; `truthy` keeps `string` / `number` / `boolean` in **both** branches (they may be falsy), drops `null` / `undefined` from the true branch, and objects from the false branch.

Example (excerpt):

```
typeof object → true: null | Date | string[] | {kind:"a"} | {kind:"b"} | false: string |
truthy → true: string | number | boolean | Date | string[] | {kind:"a"} | {kind:"b"} | f
kind === a → true: {kind:"a"} | false: string | number | boolean | null | undefined | Da
```

INPUT 1

```
string number boolean null undefined Date
string[] {kind:"a"} {kind:"b"}
typeof string
typeof object
instanceof Date
=== null
truthy
Array.isArray
in kind
kind === a
```

OUTPUT 1

```
typeof string → true: string | false:
number | boolean | null | undefined | Date
| string[] | {kind:"a"} | {kind:"b"}
typeof object → true: null | Date |
string[] | {kind:"a"} | {kind:"b"} |
false: string | number | boolean |
undefined
instanceof Date → true: Date | false:
string | number | boolean | null |
undefined | string[] | {kind:"a"} |
{kind:"b"}
=== null → true: null | false: string |
number | boolean | undefined | Date |
string[] | {kind:"a"} | {kind:"b"}
truthy → true: string | number | boolean
| Date | string[] | {kind:"a"} |
{kind:"b"} | false: string | number |
boolean | null | undefined
Array.isArray → true: string[] | false:
string | number | boolean | null |
undefined | Date | {kind:"a"} | {kind:"b"}
in kind → true: {kind:"a"} | {kind:"b"} |
false: string | number | boolean | null |
undefined | Date | string[]
kind === a → true: {kind:"a"} | false:
string | number | boolean | null |
undefined | Date | string[] | {kind:"b"}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
// Union members: string number boolean null undefined Date string[] {kind:"a"} {kind:"b"
const members = lines[0].trim().split(/\s+/);
function describe(m) {
  if (m === "null") return { typeof: "object", isNull: true, isDate: false, isArray: fals
  if (m === "undefined") return { typeof: "undefined", isNull: false, isDate: false, isAr
  if (m === "string" || m === "number" || m === "boolean") return { typeof: m, isNull: fa
  if (m === "Date") return { typeof: "object", isNull: false, isDate: true, isArray: fals
  if (m.endsWith("[]")) return { typeof: "object", isNull: false, isDate: false, isArray:
  const kind = /kind:"([^"]+)"/.exec(m)?.[1] ?? null;
  return { typeof: "object", isNull: false, isDate: false, isArray: false, kind, canBeFal
}
function narrow(members, check) {
  const yes = [], no = [];
  for (const m of members) {
    const d = describe(m);
    let result;                                  // true = definitely passes, false = definit
    const [op, ...rest] = check.split(/\s+/);
    const arg = op === "kind" ? rest[1] : rest[0];       // `kind === a` carries the oper
    if (op === "typeof") result = d.typeof === arg;
    else if (op === "instanceof") result = d.isDate;
    else if (op === "===") result = d.isNull;
    else if (op === "truthy") result = d.alwaysFalsy ? false : d.canBeFalsy ? "both" : tr
    else if (op === "Array.isArray") result = d.isArray;
    else if (op === "in") result = d.kind !== null;
    else if (op === "kind") result = d.kind === null ? false : d.kind === arg;
    else throw new SyntaxError(`unknown check '${check}'`);
    if (result === true || result === "both") yes.push(m);
    if (result === false || result === "both") no.push(m);     // a falsy-capable primiti
  }
  return { yes, no };
}
const show = (ms) => ms.join(" | ") || "never";
for (const line of lines.slice(1)) console.log(`${line.trim()} → true: ${show(narrow(mem
```

EXERCISE 2

Type guards and assertion functions

Write `isUser(x)` — the runtime body of a guard `x is User` for `{ id: string; name: string; email?: string }` (name non-empty) — and `assertIsUser(x)` (`asserts x is User`) that throws `TypeError("not a User: <reasons joined by , >")`, with reasons `expected an object, got <null|array|typeof>`, `id must be a string`, `name must be a non-empty string`, `email must be a string when present`. For each JSON line print `User <name>` (plus `<email>` when present) or the error message. Then filter all values with the guard (`users.filter(isUser)` — typed `User[]` in TypeScript) and print `users=<n> via filter: <ids or →>`, and finally `assert(users.length > 0, "at least one user required")` printing `first user id=<id>` or `assertion failed: <message>`.

Example (excerpt):

```
User Ada <ada@x.io>
not a User: id must be a string
not a User: name must be a non-empty string
not a User: expected an object, got string
users=2 via filter: 1,5
first user id=1
```

INPUT 1

```
{"id":"1","name":"Ada","email":"ada@x.io"}
{"id":2,"name":"Bob"}
{"id":"3","name":""}
"just a string"
{"id":"4","name":"Cy","email":42}
null
{"id":"5","name":"Dee"}
```

OUTPUT 1

```
User Ada <ada@x.io>
not a User: id must be a string
not a User: name must be a non-empty
string
not a User: expected an object, got string
not a User: email must be a string when
present
not a User: expected an object, got null
User Dee
users=2 via filter: 1,5
first user id=1
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
function userProblems(x) {
  if (x === null || typeof x !== "object" || Array.isArray(x)) return [`expected an objec
  const problems = [];
  if (typeof x.id !== "string") problems.push("id must be a string");
  if (typeof x.name !== "string" || x.name.trim() === "") problems.push("name must be a n
  if (x.email !== undefined && typeof x.email !== "string") problems.push("email must be
  return problems;
}
function isUser(x) /* : x is User */ { return userProblems(x).length === 0; }          //
function assertIsUser(x) /* : asserts x is User */ {
  const problems = userProblems(x);
  if (problems.length) throw new TypeError(`not a User: ${problems.join(", ")}`);
}
function assert(cond, msg) /* : asserts cond */ { if (!cond) throw new Error(msg); }
const values = lines.map((l) => { try { return JSON.parse(l); } catch { return Symbol.for
for (const v of values) {
  if (v === Symbol.for("invalid")) { console.log("invalid JSON"); continue; }
  try { assertIsUser(v); console.log(`User ${v.name}${v.email ? ` <${v.email}>` : ""}`);
  catch (err) { console.log(err.message); }
}
const users = values.filter(isUser);                  // with a guard, TypeScript types th
console.log(`users=${users.length} via filter: ${users.map((u) => u.id).join(",") || "-"}
try { assert(users.length > 0, "at least one user required"); console.log(`first user id=
catch (err) { console.log(`assertion failed: ${err.message}`); }
```

QUIZ

1. `if (typeof x === "object") x.foo` on `x: object | null` fails because…

   A. Objects have no properties

   B. `typeof null` is `"object"`, so `null` survives the check — test `x !== null` too

   C. `typeof` does not narrow

   D. `foo` is misspelled

2. A function declared `function isUser(x: unknown): x is User` …

   A. Is checked by the compiler for correctness

   B. Narrows `x` to `User` where it returns true — its body is ordinary JavaScript that **you** must make correct

   C. Converts `x` to a User

   D. Throws on failure

3. `const user = JSON.parse(text) as User` …

   A. Validates the JSON

   B. Performs no check at all — a promise the compiler cannot verify; validate with a guard or schema instead

   C. Throws on mismatch

   D. Is the recommended pattern

4. `default: { const x: never = value; … }` in a switch over a union…

   A. Is dead code

   B. Fails to compile when a union member is not handled — exhaustiveness checking

   C. Runs for every case

   D. Is a runtime check only

5. Narrowing is lost after…

   A. Nothing — it persists

   B. A function call, an `await` or an assignment that could have changed the value — extract to a `const` to keep it

   C. A semicolon

   D. A comment

ANSWER KEY · NARROWING, TYPE GUARDS, ASSERTIONS — MAKING THE COMPILER TRUST A VALUE

1. B Truthiness narrowing has the mirror trap: it drops `0` and `""`.

2. B `asserts x is User` is the throwing variant that narrows for the rest of the scope.

3. B `as` and `!` compile to nothing.

4. B At run time the branch also catches data the types never anticipated.

5. B Control-flow analysis resets to the declared type when it cannot prove the value unchanged.

LESSON 5 · 12 MIN

TypeScript in practice — tsconfig, declaration files, builds, migration and the errors you will see

Knowing the type system is half of using TypeScript; the other half is the toolchain around it — a `tsconfig.json` whose dozen important options decide how strict and how compatible your build is, declaration files that describe JavaScript to the compiler, a build step that has to fit next to bundlers and test runners, and a migration path for existing JavaScript. This lesson is the practical layer: the options that matter and why, `.d.ts` files and `@types`, type-only imports, `tsc` versus fast transpilers, the migration recipe, runtime validation as the type system's partner, and the compiler errors you will meet most, decoded.

`tsconfig.json`, the options that matter

```
{
  "compilerOptions": {
    "strict": true,                        // the whole strict family — non-negot
    "target": "ES2022",                    // the JS syntax to emit; match your r
    "module": "NodeNext",                  // how imports are emitted/resolved: N
    "moduleResolution": "NodeNext",        // (or "Bundler" for Vite/webpack proj
    "lib": ["ES2022", "DOM"],              // built-in type libraries; drop DOM f
    "esModuleInterop": true,               // sane default imports from CommonJS
    "noUncheckedIndexedAccess": true,      // arr[i] is T | undefined — catches t
    "exactOptionalPropertyTypes": true,    // optional means absent, not "or unde
    "noImplicitOverride": true, "noFallthroughCasesInSwitch": true,
    "skipLibCheck": true,                  // do not re-check node_modules .d.ts
    "outDir": "dist", "rootDir": "src", "declaration": true, "sourceMap": true,
    "isolatedModules": true                // each file transpilable alone — requ
  },
  "include": ["src"]
}
```

`strict` enables `noImplicitAny`, `strictNullChecks` (the big one — `null` / `undefined` are not in every type), `strictFunctionTypes`, `strictPropertyInitialization`, `useUnknownInCatchVariables` and more. Turning `strict` off to make errors disappear removes most of TypeScript's value; turn *on* the extra checks above as the project matures. `target` decides which syntax is downlevelled (`??`, `?.`, class fields) — set it to what your Node/browsers support so output stays readable and fast. `module` / `moduleResolution` mismatches produce the `Cannot use import statement` / `ERR_REQUIRE_ESM` family at run time (module 9): for Node projects use `NodeNext` and let `package.json` `"type"` decide.

Declaration files and `@types`

A `.d.ts` file declares types with no implementation — how the compiler knows about `fs`, the DOM, and every JavaScript library:

```
// types/legacy-lib.d.ts
declare module "legacy-lib" {
  export function parse(input: string, opts?: { strict?: boolean }): Record<string
}
declare global { interface Window { analytics?: { track(event: string): void } } }
```

Libraries ship their own (`"types"` in package.json) or the community provides them in **DefinitelyTyped** (`npm i -D @types/express`). Missing types → `Could not find a declaration file for module 'x'` → install `@types/x`, or write a minimal `declare module "x";` (everything

`any` — a stopgap). `tsc --declaration` emits `.d.ts` for your own package so consumers get types without your source. `declare` means "this exists at run time; here is its type" — variables, functions, modules, globals.

Type-only imports and erasure

```
import type { User } from "./types";          // erased entirely — no runtime impo
import { type Config, loadConfig } from "./config";   // mixed: Config erased, loa
export type { User };
```

With `isolatedModules` / `verbatimModuleSyntax`, imports used only as types must say so, or the transpiler cannot know whether to keep the import (a runtime import of a types-only module would fail or run side effects). Habit: `import type` for anything that is only a type.

Building and running

`tsc` type-checks **and** emits, but it is slow on large projects. The common setup separates the two: a fast transpiler strips types for running and bundling — **esbuild**, **swc**, **tsx**/**ts-node** for scripts (this repository runs its scripts with `tsx`), Vite/Next/webpack loaders for apps, **Bun**/**Deno** natively — and `tsc --noEmit` runs the type check in the editor, in a pre-commit hook and in CI. Fast transpilers do not type-check; if CI skips `tsc --noEmit`, type errors ship. Node 22.6+ can strip types itself (`--experimental-strip-types`); still no checking.

Migrating JavaScript

- Add `tsconfig.json` with `allowJs: true`, `checkJs: true`, `strict: false`, `noEmit: true` — the checker runs over `.js` with inference and JSDoc, and reports nothing you have not asked for.
- Fix what `checkJs` finds; add JSDoc types at module boundaries.
- Rename files to `.ts` leaf-first (utilities before the code that imports them); `any` where needed with a `// TODO(types)` comment.
- Turn on `strict` flags one at a time (`noImplicitAny`, then `strictNullChecks` — the largest batch).
- Delete the `any`s. Track their count; it should only go down.

Big-bang rewrites fail; the incremental path works because TypeScript accepts a mixed codebase.

Types and runtime validation

Types vanish at run time, so anything from outside needs a check the compiler can rely on. A schema library defines the shape **once** and provides both:

```
import { z } from "zod";
const User = z.object({ id: z.string(), name: z.string().min(1), email: z.string()
type User = z.infer<typeof User>;                  // the static type, derived fro
const user = User.parse(await res.json());          // runtime validation → a type
```

Hand-written guards (lesson 4) scale to a few shapes; schemas scale to an API. Either way the rule is the same: validate at the boundary, then trust the types inside.

The errors you will see most

Error	Means	Fix
`TS2322: Type 'X' is not assignable to type 'Y'`	shapes differ; read the *last* line of the message for the exact property	fix the value or the type; do not `as`
`TS2339: Property 'x' does not exist on type 'Y'`	typo, missing narrowing, or the type is a union without `x` on every member	narrow first; check spelling
`TS2345: Argument of type 'A' is not assignable to parameter of type 'B'`	wrong argument shape	same as 2322
`TS7006: Parameter 'x' implicitly has an 'any' type`	un-annotated parameter under `noImplicitAny`	annotate it
`TS2531/TS18047/TS18048: Object is possibly 'null' / 'undefined'`	`strictNullChecks` — the value may be absent	check, `?.`, `??`, or handle the case; `!` only when certain
`TS2307: Cannot find module 'x' or its corresponding type declarations`	not installed, wrong path/extension, or missing `@types`	install; fix the path; add a declaration
`TS2564: Property 'x' has no initializer and is not definitely assigned`	class field not set in the constructor	initialise, make optional, or `!` when a framework assigns it
`TS2352: Conversion of type 'A' to type 'B' may be a mistake`	`as` between unrelated types	fix the logic; `as unknown as B` only knowingly
`TS1259/TS1192: Module has no default export`	CJS/ESM interop	`esModuleInterop`, or `import * as x`
`TS2589: Type instantiation is excessively deep`	a recursive type exploded	simplify the type

Read errors bottom-up: the last "Type 'X' is not assignable to type 'Y'" line names the actual mismatch.

Common mistakes

- `strict: false` in a new project; `skipLibCheck` off (slow) or `noEmit` misunderstood.
- Running only a transpiler (esbuild/tsx) with no `tsc --noEmit` in CI — errors ship.
- `import` of a type without `type` under isolated modules; `const enum` in bundled projects.
- `as User` on parsed JSON instead of a schema/guard.
- Migrating by rewriting everything at once; leaving `any` uncounted.
- Ignoring the last line of a long error message.

Interview angle

"Which tsconfig options matter most?" `strict`, `target`, `module` / `moduleResolution`, `esModuleInterop`, `noUncheckedIndexedAccess`, `skipLibCheck`, `isolatedModules`.

"What is a `.d.ts` file?" Type declarations without implementation — how the compiler knows JavaScript libraries and globals; `@types/*` from DefinitelyTyped.

"How do you run TypeScript in production?" Transpile with tsc/esbuild/swc to JavaScript; type-check separately with `tsc --noEmit` in CI.

"How would you migrate a JavaScript codebase?" `allowJs` + `checkJs`, JSDoc at boundaries, rename leaf-first, enable strict flags incrementally, drive `any` to zero.

"How do types relate to runtime validation?" Types are erased; validate at boundaries with guards or a schema library and infer the type from the schema.

Key takeaways

`strict: true` always; match `target` to the runtime; `NodeNext` for Node; `noUncheckedIndexedAccess`, `skipLibCheck`, `isolatedModules` on.

`.d.ts` and `@types` describe JavaScript to the compiler; `declare module` / `declare global` for gaps; `import type` for type-only imports.

Transpile fast (esbuild/swc/tsx), type-check separately (`tsc --noEmit`) in editor, hook and CI.

Migrate incrementally: `checkJs` → JSDoc → rename leaf-first → strict flags → zero `any`.

Validate at boundaries with schemas whose types you infer; read compiler errors from the last line.

EXERCISE 1

A tsconfig doctor

Read a `tsconfig.json` and print findings, each on its own line as `- <error|warn|recommend|info>: <message>`, preceded by `tsconfig: <n> finding(s)` (and `- looks good` when none). Rules, in order: `strict` not true → error (`strict is off - enable it (implies noImplicitAny, strictNullChecks, strictFunctionTypes, useUnknownInCatchVariables)`); missing `noUncheckedIndexedAccess`, `skipLibCheck`, `isolatedModules` → one recommend each; no `target` → warn; a target of ES3–ES2019 → warn (`target <t> is older than the runtime supports - Node 16 runs ES2021`); `module` / `moduleResolution` where exactly one is NodeNext/Node16 → warn that they disagree; `module: commonjs` without `esModuleInterop` → warn; no `include` / `files` → warn; `outDir` without `rootDir` → recommend; `noEmit` with `outDir` → info.

Example: `{"compilerOptions": {"strict":false,"target":"es5","module":"commonjs","moduleResolution":"nodenext","outDir":"dist"}}` → 9 findings, beginning

```
tsconfig: 9 finding(s)
- error: strict is off - enable it (implies noImplicitAny, strictNullChecks, strictFuncti
- recommend: noUncheckedIndexedAccess: arr[i] becomes T | undefined and catches the most
```

INPUT 1

```
{"compilerOptions":
{"strict":false,"target":"es5","module":"commonjs","moduleResolution":"nodenext","outDir":"dist"}}
```

OUTPUT 1

```
tsconfig: 
- error: s
enable it
noImplicit
strictNull
strictFunc
useUnknown
- recommen
noUnchecke
arr[i] bec
undefined 
most commo
- recommen
do not re-
node_modul
files
- recommen
isolatedMo
for esbuil
transpiler
- warn: ta
than the r
Node 16 ru
- warn: mo
moduleReso
disagree
- warn: co
esModuleIn
imports fr
misbehave
- warn: no
every .ts 
is compile
aside
- recommen
alongside 
output lay
predictabl
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
const config = JSON.parse(input);
const opts = config.compilerOptions ?? {};
const findings = [];
const level = (sev, msg) => findings.push(`${sev}: ${msg}`);
if (opts.strict !== true) level("error", "strict is off - enable it (implies noImplicitAr
if (opts.noUncheckedIndexedAccess !== true) level("recommend", "noUncheckedIndexedAccess:
if (opts.skipLibCheck !== true) level("recommend", "skipLibCheck: do not re-check node_mo
if (opts.isolatedModules !== true) level("recommend", "isolatedModules: required for esbu
const targetYear = /^es(\d{4})$/i.exec(String(opts.target ?? "es5"))?.[1];
if (!opts.target) level("warn", "no target - defaults to ES5 and downlevels everything");
else if (/^es(3|5|6|2015|2016|2017|2018|2019)$/i.test(opts.target)) level("warn", `target
const mod = String(opts.module ?? "").toLowerCase(), res = String(opts.moduleResolution 
if ((mod === "nodenext" || mod === "node16") !== (res === "nodenext" || res === "node16")
if (mod === "commonjs" && opts.esModuleInterop !== true) level("warn", "commonjs without 
if (!config.include && !config.files) level("warn", "no include/files - every .ts under t
if (opts.outDir && !opts.rootDir) level("recommend", "set rootDir alongside outDir so the
if (opts.noEmit === true && opts.outDir) level("info", "noEmit with outDir: this config c
console.log(`tsconfig: ${findings.length} finding(s)`);
for (const f of findings) console.log(`- ${f}`);
if (!findings.length) console.log("- looks good");
```

EXERCISE 2

Decode the compiler's errors

Each line is a TypeScript diagnostic beginning with its code. Build a rule table and print `<code>: <hint>`: TS2322 (`a '<X>' was given where a '<Y>' is expected - fix the value or the type, do not cast`), TS2345 (`the argument is a '<A>' but the parameter wants '<B>'`), TS2339 (`no '<p>' on '<T>' - a typo, or narrow the union first`), TS7006 (`annotate parameter '<x>' (noImplicitAny)`), TS2531/TS18047/TS18048 (`'<expr>' may be <null|undefined> - check it, use ?. / ??, or handle the case (strictNullChecks)`, with `the value` when no expression is named), TS2307 (`install '<m>' (and its @types), or fix the path/extension, or add a declaration`), TS2564 (`initialise '<p>' in the constructor, make it optional, or use ! when a framework assigns it`), TS2352 (`'<A>' and '<B>' do not overlap - the 'as' is hiding a logic error`), TS1259 (`enable esModuleInterop or use import * as x from '<m>'`); anything else → `no rule - read the last 'is not assignable' line of the message for the real mismatch`.

Example (excerpt):

```
TS2339: no 'nmae' on 'User' - a typo, or narrow the union first
TS18048: 'user.email' may be undefined - check it, use ?. / ??, or handle the case (strict
TS2589: no rule - read the last 'is not assignable' line of the message for the real mism
```

INPUT 1

```
TS2322: Type 'string' is not assignable to
type 'number'.
TS2339: Property 'nmae' does not exist on
type 'User'.
TS2345: Argument of type '"zip"' is not
assignable to parameter of type 'keyof
User'.
TS7006: Parameter 'x' implicitly has an
'any' type.
TS18048: 'user.email' is possibly
'undefined'.
TS2531: Object is possibly 'null'.
TS2307: Cannot find module 'lodahs' or its
corresponding type declarations.
TS2564: Property 'db' has no initializer
and is not definitely assigned in the
constructor.
TS2352: Conversion of type 'string' to
type 'number' may be a mistake because
neither type sufficiently overlaps with
the other.
TS2589: Type instantiation is excessively
deep and possibly infinite.
```

OUTPUT 1

```
TS2322: a 'string' was given where a
'number' is expected - fix the value or
the type, do not cast
TS2339: no 'nmae' on 'User' - a typo, or
narrow the union first
TS2345: the argument is a '"zip"' but the
parameter wants 'keyof User'
TS7006: annotate parameter 'x'
(noImplicitAny)
TS18048: 'user.email' may be undefined -
check it, use ?. / ??, or handle the case
(strictNullChecks)
TS2531: the value may be null - check it,
use ?. / ??, or handle the case
(strictNullChecks)
TS2307: install 'lodahs' (and its @types),
or fix the path/extension, or add a
declaration
TS2564: initialise 'db' in the
constructor, make it optional, or use !
when a framework assigns it
TS2352: 'string' and 'number' do not
overlap - the 'as' is hiding a logic error
TS2589: no rule - read the last 'is not
assignable' line of the message for the
real mismatch
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
const rules = [
  [/^TS2322: Type '(.+?)' is not assignable to type '(.+?)'/, (m) => `a '${m[1]}' was giv
  [/^TS2345: Argument of type '(.+?)' is not assignable to parameter of type '(.+?)'/, (m
  [/^TS2339: Property '(.+?)' does not exist on type '(.+?)'/, (m) => `no '${m[1]}' on '$
  [/^TS7006: Parameter '(.+?)' implicitly has an 'any' type/, (m) => `annotate parameter
  [/^TS(?:2531|18047|18048): (?:Object is possibly|'(.+?)' is possibly) '(null|undefined)
  [/^TS2307: Cannot find module '(.+?)'/, (m) => `install '${m[1]}' (and its @types), or
  [/^TS2564: Property '(.+?)' has no initializer/, (m) => `initialise '${m[1]}' in the co
  [/^TS2352: Conversion of type '(.+?)' to type '(.+?)' may be a mistake/, (m) => `'${m[1
  [/^TS1259: Module '(.+?)' can only be default-imported using the 'esModuleInterop' flag
];
for (const line of lines) {
  const text = line.trim();
  const code = /^TS\d+/.exec(text)?.[0] ?? "TS????";
  const hit = rules.map(([re, hint]) => { const m = re.exec(text); return m && hint(m); }
  console.log(`${code}: ${hit ?? "no rule - read the last 'is not assignable' line of the
}
```

QUIZ

1. `"strict": true` in tsconfig…

   A. Only forbids `any`

   B. Turns on the whole strict family — `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes`, `useUnknownInCatchVariables` and more

   C. Slows compilation only

   D. Is the default

2. A `.d.ts` file contains…

   A. Compiled JavaScript

   B. Type declarations without implementation — how the compiler knows about libraries and globals (`@types/*`, `declare module`)

   C. Tests

   D. Source maps

3. A project built with esbuild but without `tsc --noEmit` in CI…

   A. Is fully checked

   B. Ships type errors — esbuild strips types without checking them

   C. Cannot build

   D. Is slower

4. The recommended way to migrate a JavaScript codebase is…

   A. Rewrite everything as `.ts` at once

   B. Incrementally: `allowJs` / `checkJs` with JSDoc, rename leaf files first, enable strict flags one at a time, drive `any` to zero

   C. Add `any` everywhere

   D. Start with `strict: true` on all files

5. `z.infer<typeof UserSchema>`…

   A. Validates at run time

   B. Derives the static type from a runtime schema, so one definition gives both validation and the type

   C. Is a type guard

   D. Compiles the schema

ANSWER KEY · TYPESCRIPT IN PRACTICE — TSCONFIG, DECLARATION FILES, BUILDS, MIGRATION AND THE ERRORS YOU WILL SEE

1. B Turning it off to silence errors removes most of TypeScript's value.

2. B `tsc --declaration` emits them for your own package.

3. B Transpile fast, type-check separately.

4. B TypeScript accepts a mixed codebase precisely for this.

5. B The bridge between erased types and data from outside.

CHECKPOINT · 24 MIN

Checkpoint — TypeScript preview

This checkpoint covers erased types and inference, literal types and widening, `any` / `unknown` / `never`; object types, `interface` versus `type`, structural typing and excess-property checks, intersections, discriminated unions and `satisfies`; function types, generics with constraints, `keyof`/indexed access and the utility types; narrowing, exhaustiveness, type guards, assertion functions and the escape hatches; and the toolchain — tsconfig, declaration files, builds, migration, runtime validation.

How it works. Twelve questions and three programs; 70% on the questions and every program accepted clears the module. The programs are JavaScript (the judge runs Node) that model the type system's own rules — inference and widening, structural assignability, utility types over schemas, narrowing — so you reason about what the compiler would do.

Before you start, make sure you can answer:

- What `const x = "a"` versus `let x = "a"` infers; what `as const` does; the difference between `any`, `unknown` and `never`.
- Why a class name does not matter for assignability; when excess properties are an error; what a discriminated union's tag is for.
- What `K extends keyof T` and `T[K]` express; how `Omit` is built; what a mapped type is.
- Which checks narrow; why `typeof x === "object"` keeps `null`; what `x is T` and `asserts x is T` do; when `as` is acceptable.
- What `strict` turns on; what a `.d.ts` is; why `tsc --noEmit` must run in CI when esbuild builds; how to migrate JavaScript.

The programs are a schema builder that both validates values and prints the TypeScript type it would infer, an exhaustive state machine over discriminated-union states and events, and a declaration-file generator that infers `.d.ts` text from runtime values.

EXERCISE 1

A schema builder that validates and infers

Line 1 is a schema DSL: space-separated `key:type` fields where a trailing `?` on the key means optional and a type is `string`, `number`, `boolean`, a `T[]` array, or a union of string literals `"admin"|"user"`. Compile it, print `type = { key: type; key?: type; … }` in TypeScript syntax (a literal union in an array is parenthesised), then `parse` each following JSON line collecting **every** problem: `<key>: required`, `<path>: expected <type>, got <string|number|boolean|null|array>`, `<path>: expected <"a" | "b">, got <JSON>`, with array items addressed as `tags[1]`; a non-object root gives `<root>: expected an object`. Print `ok <JSON>` or `error <problems joined by " | ">`.

Example: `name:string age:number email?:string tags:string[] role:"admin"|"user"` →

```
type = { name: string; age: number; email?: string; tags: string[]; role: "admin" | "user
ok {"name":"Ada","age":36,"tags":["x"],"role":"admin"}
error age: expected number, got string | tags[1]: expected string, got number | role: exp
error name: required
error <root>: expected an object
```

INPUT 1

```
name:string age:number email?:string
tags:string[] role:"admin"|"user"
{"name":"Ada","age":36,"tags":
["x"],"role":"admin"}
{"name":"Bob","age":"17","tags":
["x",1],"role":"guest"}
{"age":1,"tags":[],"role":"user"}
[1]
{"name":"Cy","age":2,"email":"c@x","tags":
[],"role":"user"}
```

OUTPUT 1

```
type = { name: string; age: number;
email?: string; tags: string[]; role:
"admin" | "user" }
ok {"name":"Ada","age":36,"tags":
["x"],"role":"admin"}
error age: expected number, got string |
tags[1]: expected string, got number |
role: expected "admin" | "user", got
"guest"
error name: required
error <root>: expected an object
ok
{"name":"Cy","age":2,"email":"c@x","tags":
[],"role":"user"}
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =>
// Schema DSL on line 1:  name:string age:number email?:string tags:string[] role:"admin"
function compileType(text) {
  if (text.endsWith("[]")) return { kind: "array", of: compileType(text.slice(0, -2)) };
  if (text.startsWith("\"")) return { kind: "literal", values: text.split("|").map((v) =>
  if (["string", "number", "boolean"].includes(text)) return { kind: text };
  throw new SyntaxError(`unknown type '${text}'`);
}
function compile(dsl) {
  return dsl.trim().split(/\s+/).map((field) => {
    const [rawKey, type] = field.split(":");
    return { key: rawKey.replace(/\?$/, ""), optional: rawKey.endsWith("?"), type: compil
  });
}
function typeText(t) {
  if (t.kind === "array") return `${t.of.kind === "literal" ? `(${typeText(t.of)})` : typ
  if (t.kind === "literal") return t.values.map((v) => JSON.stringify(v)).join(" | ");
  return t.kind;
}
const toType = (schema) => `{ ${schema.map((f) => `${f.key}${f.optional ? "?" : ""}: ${ty
function checkType(value, t, path, errors) {
  if (t.kind === "array") { if (!Array.isArray(value)) errors.push(`${path}: expected ${t
  if (t.kind === "literal") { if (!t.values.includes(value)) errors.push(`${path}: expect
  if (typeof value !== t.kind) errors.push(`${path}: expected ${t.kind}, got ${value ===
}
function parse(schema, value) {
  const errors = [];
  if (value === null || typeof value !== "object" || Array.isArray(value)) return { ok: f
  for (const f of schema) {
    if (!(f.key in value)) { if (!f.optional) errors.push(`${f.key}: required`); continue
    checkType(value[f.key], f.type, f.key, errors);
  }
  return errors.length ? { ok: false, errors } : { ok: true, value };
}
const schema = compile(lines[0]);
console.log(`type = ${toType(schema)}`);
for (const line of lines.slice(1)) {
  let value;
  try { value = JSON.parse(line); } catch { console.log("error <root>: invalid JSON"); co
  const r = parse(schema, value);
  console.log(r.ok ? `ok ${JSON.stringify(r.value)}` : `error ${r.errors.join(" | ")}`);
}
```

EXERCISE 2

An exhaustive state machine

States: `{ status: "idle" }` | `{ status: "loading", attempt }` | `{ status: "success", data }` | `{ status: "failure", error, attempt }`. Events: `FETCH`, `RESOLVE <data>`, `REJECT <error>`, `RETRY`,

`RESET`. Write `transition(state, event)` switching on `event.type`: FETCH from idle → loading(1); RESOLVE from loading → success; REJECT from loading → failure (keeping the attempt); RETRY from failure → loading(attempt + 1); RESET → idle from anywhere; any other combination returns the **same** state object; an unknown event type reaches `assertNever` (`unhandled: <JSON of the event>`). Print `<TYPE>: <before> → <after>` with `(ignored)` when the state object did not change, or `<TYPE>: <error message>`; describe states as `idle`, `loading(attempt n)`, `success(<JSON data>)`, `failure(<error>, attempt n)`.

Example: `FETCH`, `FETCH`, `REJECT timeout`, `RETRY`, `RESOLVE {"items":3}`, `RESET`, `EXPLODE` →

```
FETCH: idle → loading(attempt 1)
FETCH: loading(attempt 1) → loading(attempt 1) (ignored)
REJECT: loading(attempt 1) → failure(timeout, attempt 1)
RETRY: failure(timeout, attempt 1) → loading(attempt 2)
RESOLVE: loading(attempt 2) → success("{\"items\":3}")
RESET: success("{\"items\":3}") → idle
EXPLODE: unhandled: {"type":"EXPLODE"}
```

INPUT 1

```
FETCH
FETCH
REJECT timeout
RETRY
RESOLVE {"items":3}
RETRY
RESET
EXPLODE
RESOLVE x
```

OUTPUT 1

```
FETCH: idle → loading(attempt 1)
FETCH: loading(attempt 1) →
loading(attempt 1) (ignored)
REJECT: loading(attempt 1) →
failure(timeout, attempt 1)
RETRY: failure(timeout, attempt 1) →
loading(attempt 2)
RESOLVE: loading(attempt 2) → success("
{\"items\":3}")
RETRY: success("{\"items\":3}") →
success("{\"items\":3}") (ignored)
RESET: success("{\"items\":3}") → idle
EXPLODE: unhandled: {"type":"EXPLODE"}
RESOLVE: idle → idle (ignored)
```

REFERENCE SOLUTION

```
"use strict";
const lines = require("fs").readFileSync(0, "utf8").split("\n").filter((l) => l.trim() =
// State: { status: "idle" } | { status: "loading"; attempt } | { status: "success"; data
// Event: FETCH | RESOLVE <data> | REJECT <error> | RETRY | RESET
function assertNever(x) { throw new Error(`unhandled: ${JSON.stringify(x)}`); }
function transition(state, event) {
  switch (event.type) {
    case "FETCH": return state.status === "idle" ? { status: "loading", attempt: 1 } : st
    case "RESOLVE": return state.status === "loading" ? { status: "success", data: event.
    case "REJECT": return state.status === "loading" ? { status: "failure", error: event.
    case "RETRY": return state.status === "failure" ? { status: "loading", attempt: state
    case "RESET": return { status: "idle" };
    default: return assertNever(event);          // an event the union does not know
  }
}
const describe = (s) => s.status === "idle" ? "idle" : s.status === "loading" ? `loading(
let state = { status: "idle" };
for (const line of lines) {
  const [type, ...rest] = line.trim().split(/\s+/);
  const event = type === "RESOLVE" ? { type, data: rest.join(" ") } : type === "REJECT" ?
  try {
    const next = transition(state, event);
    console.log(`${type}: ${describe(state)} → ${describe(next)}${next === state ? " (ig
    state = next;
  } catch (err) {
    console.log(`${type}: ${err.message}`);
  }
}
```

EXERCISE 3

Generate a declaration file

The input is a JSON object describing a module's exports: a primitive, array or plain-object value is a constant; `{"fn": <arity>}` is a function; `{"class": [fields], "methods": [names]}` is a class. Infer types the way a declaration emitter would — `string`, `number`, `boolean`, `null`, arrays as `T[]` or `(A | B)[]` in first-seen order (`unknown[]` when empty), objects as `{ k: T; … }` — and print one declaration per export in key order: `export declare const NAME: T;`, `export declare function f(arg0: unknown, arg1: unknown): unknown;`, and multi-line `export declare class C {` with `field: unknown;` and `method(): unknown;` lines and `}` (an empty class prints `{}` on one line).

Example (excerpt):

```
export declare const VERSION: string;
export declare const flags: { debug: boolean; level: string };
export declare const mixed: (number | string | null)[];
export declare function add(arg0: unknown, arg1: unknown): unknown;
export declare class Point {
  x: unknown;
  y: unknown;
  dist(): unknown;
  toString(): unknown;
}
export declare class Empty {}
```

INPUT 1

```
{"VERSION":"1.2.3","MAX":10,"flags":
{"debug":false,"level":"info"},"names":
["a","b"],"mixed":[1,"a",null],"empty":
[],"add":{"fn":2},"noop":{"fn":0},"Point":
{"class":["x","y"],"methods":
["dist","toString"]},"Empty":{"class":[]}}
```

OUTPUT 1

```
export declare const VERSION: string;
export declare const MAX: number;
export declare const flags: { debug:
boolean; level: string };
export declare const names: string[];
export declare const mixed: (number |
string | null)[];
export declare const empty: unknown[];
export declare function add(arg0: unknown,
arg1: unknown): unknown;
export declare function noop(): unknown;
export declare class Point {
  x: unknown;
  y: unknown;
  dist(): unknown;
  toString(): unknown;
}
export declare class Empty {}
```

REFERENCE SOLUTION

```
"use strict";
const input = require("fs").readFileSync(0, "utf8");
// Exports described as JSON: primitives/arrays/objects are values; {"fn": <arity>} is a
const exportsMap = JSON.parse(input);
function inferType(value) {
  if (value === null) return "null";
  if (Array.isArray(value)) {
    const members = [...new Set(value.map(inferType))];
    return members.length === 0 ? "unknown[]" : members.length === 1 ? `${members[0]}[]`
  }
  if (typeof value === "object") return `{ ${Object.entries(value).map(([k, v]) => `${k}:
  return typeof value;
}
const out = [];
for (const [name, value] of Object.entries(exportsMap)) {
  if (value && typeof value === "object" && !Array.isArray(value) && typeof value.fn ===
    const params = Array.from({ length: value.fn }, (_, i) => `arg${i}: unknown`).join(",
    out.push(`export declare function ${name}(${params}): unknown;`);
  } else if (value && typeof value === "object" && Array.isArray(value.class)) {
    const fields = value.class.map((f) => `  ${f}: unknown;`);
    const methods = (value.methods ?? []).map((m) => `  ${m}(): unknown;`);
    const members = [...fields, ...methods];
    out.push(members.length ? `export declare class ${name} {\n${members.join("\n")}\n}`
  } else {
    out.push(`export declare const ${name}: ${inferType(value)};`);
  }
}
console.log(out.join("\n"));
```

CHECKPOINT QUESTIONS

1. `let x = 5` then `x = "a"` …

A. Compiles — `let` is dynamic

B. Fails — `x` was inferred as `number` from its initialiser

C. Compiles with a warning

D. Makes `x` `any`

2. `string & number` is…

A. `string | number`

B. `never` — no value is both; conflicting intersections collapse

C. `any`

D. An error

3. Excess-property checking applies when…

A. Always

B. An object **literal** is written directly where a type is expected — the same object via a variable is accepted

C. Using `interface` only

D. Never

4. `import type { User } from "./types"` …

A. Imports the runtime value

B. Is erased entirely — required by isolated-module transpilers when the import is only used as a type

C. Is slower

D. Is a syntax error

5. `Record<"GET" | "POST", Handler>` …

A. Is an array

B. Is an object type with exactly those two keys, each a `Handler` — a typed dispatch table

C. Is a `Map`

D. Allows any string key

6. `ReturnType<typeof makeStore>` …

A. Calls `makeStore`

B. Names the type of what `makeStore` returns without writing it out — useful for types you never declared

C. Is a runtime value

D. Requires a class

7. `users.filter(isUser)` where `isUser` is a type guard is typed as…

A. `unknown[]`

B. `User[]` — a guard predicate narrows the element type; a plain boolean callback would not

C. `boolean[]`

D. `any[]`

8. `document.getElementById("q") as HTMLInputElement` is acceptable because…

A. It validates the element

B. You hold local knowledge the compiler cannot (the element's kind in your own markup) — the same `as` on JSON from a server would be a lie

C. `as` always checks

D. Elements are `any`

9. `catch (err)` under `strict` binds `err` as…

A. `Error`

B. `unknown` — narrow with `instanceof Error` (or a guard) before reading `.message`

C. `any`

D. `string`

10. `noUncheckedIndexedAccess` makes `arr[i]` …

A. Throw on out-of-range

B. `T | undefined`, forcing a check — catches the most common runtime crash at compile time

C. `never`

D. `any`

11. Reading a long `TS2322` message, the real mismatch is usually…

A. In the first line

B. In the **last** `Type 'X' is not assignable to type 'Y'` line — the message narrows down to the offending property

C. In the file name

D. In the error code

12. Types and runtime validation relate how?

A. Types validate at run time

B. Types are erased, so data from outside must be validated (guards or a schema library); inside the boundary the types are trusted

C. Validation is unnecessary with strict mode

D. Schemas replace types

ANSWER KEY · CHECKPOINT — TYPESCRIPT PREVIEW

1. B Inference assigns the widened type once; annotate `let x: number | string` if both are intended.

2. B A common source of surprise `never`s in intersected object types.

3. B It is there to catch typos, not to make types nominal.

4. B `import { type Config, loadConfig }` mixes both in one statement.

5. B `Record<string, T>` is the open-keyed dictionary form.

6. B `Parameters<F>` gives the argument tuple; `Awaited<T>` what `await` yields.

7. B One of the practical payoffs of writing real guards.

8. B Treat every `as` in a diff as a review question.

9. B `useUnknownInCatchVariables` is part of `strict`.

10. B Recommended once a project is on `strict`.

11. B Read compiler errors bottom-up.

12. B One schema definition can provide both — `z.infer` derives the static type.

CodeKairo · JavaScript study plan · Module 13: TypeScript preview. Solve the exercises online to track your progress and earn XP.