TypeScript Course

Flavio Copes

Updated August 3, 2026

Contents

Learn TypeScript from the JavaScript you already know: inference, annotations, objects, unions, functions, generics, narrowing, and compiler configuration.

What you'll learn: Add useful types to a small JavaScript program, configure the compiler, and handle uncertain data without turning type checking off.

Start using TypeScript

Install the compiler, run a first program, and read type errors.

What TypeScript adds to JavaScript

See TypeScript as JavaScript plus a static type checker, not as a separate runtime or a replacement language.

TypeScript is JavaScript plus a static type checker.

The checker studies your source code before it runs. It tracks which values can reach each operation and reports combinations that do not make sense.

For example:

```
function uppercase(name: string) {
  return name.toUpperCase()
}

uppercase(42)
```

JavaScript would try to run this call and fail inside the function. TypeScript reports the number argument before the program starts.

Types also document contracts. An editor can show that `uppercase()` expects a string and offer string methods while you type.

But TypeScript is not a new runtime. Browsers and Node.js ultimately execute JavaScript. Type annotations disappear from the emitted code, so they cannot inspect a network response or change how JavaScript behaves.

This gives us the main mental model for the course:

- TypeScript checks what the source code can prove.
- JavaScript handles the values that exist at runtime.

Exercise: change `uppercase(42)` to a valid call. Then think of one value TypeScript cannot know without a runtime check.

Install TypeScript in a project

Add the compiler as a development dependency and run the project-local version so every contributor uses the same toolchain.

Create a project folder with a `package.json`, then install TypeScript locally:

```
npm init -y
npm install --save-dev typescript
```

Check the project-local compiler:

```
npx tsc --version
```

`npx` finds the `tsc` executable from the project dependency. This keeps the expected compiler version in `package.json` instead of relying on an unrelated global installation.

That matters because compiler versions can add checks and understand new syntax. Two contributors should not get different results because their global tools differ.

The `--save-dev` flag records TypeScript as a development dependency. Your application does not need the compiler after you have produced runnable JavaScript, unless your deployment process compiles there.

If `npx tsc` cannot run, inspect `package.json` and confirm that installation completed. Do not fix a local-project problem by installing a random global version.

Exercise: run `npx tsc --version`, then find the recorded TypeScript range in `package.json`.

Write and compile your first program

Create a TypeScript file, catch one mistake, and inspect the JavaScript produced by the compiler.

Time to write some TypeScript and watch the compiler do its two jobs: check your code, then turn it into JavaScript.

Create `index.ts`:

```
const greeting: string = 'Hello'
console.log(greeting.toUpperCase())
```

The `: string` part is a type annotation. It tells TypeScript what kind of value `greeting` holds.

Compile it and run the emitted JavaScript:

```
npx tsc index.ts
node index.js
```

You should see `HELLO` printed. Notice the two-step flow: `tsc` reads the `.ts` file, `node` runs the `.js` file. Node.js never sees your TypeScript.

The compiler checks `index.ts`, removes the type annotation, and writes JavaScript. Open `index.js` and confirm that `: string` is gone. The emitted file is ordinary JavaScript, almost identical to what you wrote.

Catch your first error

Now introduce a mistake:

```
const greeting: string = 42
```

Run the compiler again. TypeScript reports that `number` is not assignable to `string`:

```
index.ts:1:7 - error TS2322: Type 'number' is not assignable to type 'string'.
```

Read the parts: the file, the line and column, an error code, and a plain-language message. You declared `greeting` as a `string`, then gave it a `number`, and the compiler caught the contradiction before the program ran.

Errors do not always block output

Here is a detail that surprises people. Check the directory after that failed compile: `index.js` was still written.

Depending on compiler settings, TypeScript can still emit JavaScript when errors exist. A type error is not automatically a runtime barrier. This is deliberate — it lets you run partially migrated code — but it means a build script that only runs `tsc` can ship broken output. Later we will use `noEmitOnError` or a separate `--noEmit` check when a workflow must stop on errors.

One more thing: you did not have to annotate everything. Delete `: string` from the working version and compile again. It still checks correctly, because TypeScript can infer the type from the value `'Hello'`. We explore inference properly in the next lessons.

Exercise: restore the string, add `const length = greeting.length`, and hover over `length` in your editor. TypeScript should infer `number` without another annotation.

Read a TypeScript error

Turn a compiler diagnostic into three concrete facts: the received type, the expected type, and the location where they conflict.

A TypeScript error usually gives you three useful facts:

1. The type TypeScript found.
2. The type the receiving code requires.
3. The source location where those types meet.

Example:

```
function setPort(port: number) {}

setPort('3000')
```

The important part of the diagnostic is: `string` is not assignable to `number`.

Start at the first error in your own source. Later errors can be consequences of that one. Hover over the argument and parameter to inspect what TypeScript inferred.

Then decide which side is wrong. If the value came from an environment variable, parse and validate it. If the function should accept text, change the contract deliberately.

Do not reach immediately for `any`, `as number`, or an ignore comment. Those silence evidence without fixing the value.

Exercise: make `setPort()` accept only numbers from 1 through 65535. Notice which part needs a type and which part needs a runtime check.

Types do not change runtime behavior

Understand that annotations are erased and cannot validate a network response, form value, or JSON file while the program runs.

This assertion tells TypeScript to trust you:

```
const user = (await response.json()) as User
```

It does not inspect the response body. The server can still return `null`, an error object, or a user with missing fields.

Types also do not convert values:

```
const port = process.env.PORT as unknown as number
```

At runtime, an environment variable is still a string. The assertion changes only what the checker believes.

Treat values from JSON, forms, storage, environment variables, and network calls as uncertain. Check them at the boundary before using a more specific type.

For a user, that means verifying the value is an object and checking the fields your program needs. A schema library can help, but ordinary JavaScript checks work too.

Exercise: log `typeof process.env.PORT` at runtime. Compare that result with what an unsafe assertion can make TypeScript claim.

Types and inference

Primitives, arrays, tuples, annotations, and special types.

Let TypeScript infer obvious types

Use inference for initialized local values and add annotations at boundaries where they make a contract clearer.

TypeScript reads initial values and infers types from them.

```
const course = 'TypeScript'
let status = 'draft'
```

Because `course` cannot be reassigned, TypeScript can keep the literal type `'TypeScript'`. The mutable `status` variable widens to `string` because another string can replace it later.

Writing `let status: string = 'draft'` repeats what TypeScript already knows.

Inference also follows expressions:

```
const lessons = 42
const label = `${lessons} lessons`
```

Hover over the values. The editor should show a numeric type for `lessons` and a string type for `label`.

Annotations are useful where the intended contract is not obvious: function inputs, exported APIs, empty collections, and values whose allowed type is wider than the initializer.

My advice is to let local implementation details infer their types. Add annotations where they communicate a decision.

Exercise: change `const course` to `let course` and inspect how its inferred type changes.

Add a type annotation when it helps

Write annotations after variable names and use them to express an intentional contract rather than restating an initializer.

Inference handles most variables, so an annotation should earn its place. There are two situations where it clearly does.

Annotations use a colon after the variable name:

```
let score: number
score = 42
```

This annotation helps because there is no initializer to inspect. Without it, TypeScript has nothing to infer from at the declaration. With it, the contract is set immediately: TypeScript now rejects a string assignment to `score`.

```
score = 'forty-two'
```

```
Type 'string' is not assignable to type 'number'.
```

Declare-then-assign shows up whenever the value comes from a branch — an `if/else` that computes the result differently per case, or a `try/catch` that assigns inside the `try`.

Widening an initial value deliberately

An annotation can also widen an initial value deliberately:

```
let status: 'draft' | 'published' = 'draft'
status = 'published'
```

Without the union, TypeScript would infer a mutable string variable. That inference is reasonable — a `let` initialized with `'draft'` usually wants other strings later — but it accepts any string, including `'pubished'` with a typo. The annotation preserves the two allowed values as a contract, so the typo fails:

```
Type '"pubished"' is not assignable to type '"draft" | "published"'.
```

Here the annotation is not restating the initializer. It records an intention inference cannot guess: this variable cycles between exactly two states.

Annotations that hurt

Skip annotations that merely repeat what the initializer already says:

```
const port: number = 3000
```

The `: number` adds no information and one more thing to update. Worse, it can discard precision — inference would give the `const` the literal type `3000`.

And do not use annotations to force a value into an incompatible type. If TypeScript rejects the

initializer, the error is information about your data. Fix the data or the contract instead of adding an assertion. An annotation that fights the compiler is a bug report you are refusing to read.

Exercise: declare `let result` without an initializer, then assign values of different kinds. Compare that with `let result: string | number` and inspect which assignments are rejected.

Use the primitive types

Describe string, number, boolean, bigint, symbol, null, and undefined values with lowercase TypeScript type names.

TypeScript uses lowercase names for JavaScript primitive values:

```
const title: string = 'TypeScript Course'
const lessons: number = 42
const published: boolean = true
```

Use `string`, `number`, and `boolean`, not `String`, `Number`, and `Boolean`. The capitalized names describe boxed wrapper objects that rarely belong in application types.

JavaScript uses one `number` type for ordinary integers and floating-point values. TypeScript does not add runtime `int` or `float` values.

Other primitives include `bigint`, `symbol`, `null`, and `undefined`. A type describes which values are allowed; it does not change their JavaScript behavior.

With strict null checking, `null` and `undefined` are not automatically accepted as strings or objects. Include them only when absence is part of the contract.

Exercise: declare one value with each lowercase primitive type. Then try replacing `string` with `String` and inspect the editor type before deciding why the lowercase form is clearer.

Type arrays

Describe a collection whose elements share a type and understand the equivalent bracket and generic syntax.

An array type answers one question: what can each element be? Both forms describe an array of strings:

```
const names: string[] = ['Ada', 'Lin']
const moreNames: Array<string> = ['Grace']
```

`string[]` and `Array<string>` mean exactly the same thing. The bracket form is shorter and more common. The generic form reads better when the element type is itself complex, like `Array<string | number>`.

The element type guards every operation

TypeScript checks values added later:

```
names.push('Margaret')
names.push(42)
```

The first call works. The second one fails with:

```
Argument of type 'number' is not assignable to parameter of type 'string'.
```

The check protects reads too. Every element you pull out of `names` is a `string`, so string methods are always safe on it.

Inference flows through array methods

Array methods receive the element type through inference:

```
const uppercaseNames = names.map(name => name.toUpperCase())
```

You did not annotate `name`, but TypeScript knows it is a `string` because `names` is a `string[]`. It also infers the result as `string[]`, since `toUpperCase()` returns a string. Chain a `.filter()` or another `.map()` after it and the types keep flowing.

Mixed elements need a union

Use a union only when mixed elements are intentional:

```
const ids: Array<string | number> = ['a1', 42]
```

Now each element is `string | number`, so reading an element gives you a value you must narrow before calling type-specific methods on it. That extra step is the honest price of a mixed collection.

One mistake to avoid

Do not widen an array to `any[]` just to make one bad insertion compile. That removes useful checking from every later read: `names[0].toUppercase()` with a typo would compile fine and crash at runtime. If TypeScript rejects a push, the element type is telling you something about your data. Fix the value or widen the union deliberately.

Exercise: create a `number[]`, map it to formatted strings, and hover over the result type.

Use a tuple for fixed positions

Describe an array whose positions have known meanings and possibly different types.

An array type says every element looks the same. Sometimes that is wrong: a coordinate pair has exactly two numbers, and position matters. A **tuple** gives each position a known meaning and type:

```
type Point = [number, number]
const origin: Point = [0, 0]
```

TypeScript rejects missing positions, extra positions in the literal, and a string where a number belongs. Try `const bad: Point = [0]` and you get:

```
Type '[number]' is not assignable to type 'Point'.
  Source has 1 element(s) but target requires 2.
```

Reading also respects positions. `origin[0]` is a `number`, and `origin[2]` is an error because the tuple has no third element.

Tuples are still JavaScript arrays at runtime. The names and position rules exist only during checking. `Array.isArray(origin)` returns `true`, and the emitted JavaScript is a plain `[0, 0]`.

Labeled positions

You can label positions for editor help:

```
type Point = [x: number, y: number]
```

Labels do not change the runtime value. It is still `[0, 0]`. But hovering over a function that accepts a `Point` now shows `x` and `y` instead of anonymous numbers, which makes call sites easier to read.

Tuples and destructuring

Tuples shine as return values, because destructuring gives the positions real names:

```
function divide(a: number, b: number): [result: number, remainder: number] {
  return [Math.floor(a / b), a % b]
}

const [result, remainder] = divide(17, 5)
// result: 4, remainder: 2
```

Both variables get the `number` type without any annotation. This convention is everywhere: React's `useState` returns a `[value, setter]` tuple.

Tuple or object?

Use tuples for compact pairs and established return conventions. Use an object when readers benefit from names at the call site:

```
type Point = { x: number; y: number }
```

With three or more positions, `point[2]` tells the reader nothing, while `point.z` explains itself. My rule: two elements with an obvious order, tuple; anything more, object.

Exercise: write a function returning `[value: string, found: boolean]`. Destructure the result and inspect the inferred types.

TypeScript: any vs unknown vs never

In TypeScript, any disables checking, unknown makes you verify before use, and never marks impossible states. Learn the differences with examples.

TypeScript gives you three special types that sound similar but behave very differently: `any`, `unknown`, and `never`. Picking the wrong one either kills type safety or blocks valid code.

any turns off checking

`any` is a catch-all. Any value fits, and TypeScript stops checking what you do with it.

We touched on this in the TypeScript introduction. My advice is to avoid `any` when you can. It removes many benefits of type checking.

Here is the danger:

```
function parseConfig(raw: any) {
  return raw.host.toUpperCase()
}

parseConfig({ host: 3000 })
```

No compile error. At runtime you get a crash because `3000` has no `toUpperCase` method.

`any` is sometimes an easy way out. But you pay for it later.

unknown is the safe alternative

`unknown` also accepts any value. The difference is you must narrow it before you use it.

This is what you want for data you do not trust yet, like JSON from an API:

```
function parseJson(text: string): unknown {
  return JSON.parse(text)
}

const data = parseJson('{"name":"Flavio"}')

// data.name // error: Object is of type 'unknown'

if (typeof data === 'object' && data !== null && 'name' in data) {
  console.log((data as { name: string }).name)
}
```

`unknown` forces you to check first. You get safety without giving up flexibility.

If you come from JavaScript, how to check types covers the runtime checks that pair well with `unknown`.

never for impossible states

`never` means a value that should not exist.

Use it when a function should not return, like one that always throws:

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

The bigger win is **exhaustive checks** in a switch:

```
type Shape =
  | { kind: 'circle', radius: number }
  | { kind: 'square', size: number }

function area(shape: Shape): number {
  switch (shape.kind) {
    case 'circle':
      return Math.PI * shape.radius ** 2
    case 'square':
      return shape.size ** 2
    default:
      const _exhaustive: never = shape
      return _exhaustive
  }
}
```

If you add a new shape to the union and forget a `case`, TypeScript errors on the `never` assignment.

The compiler tells you that you missed a branch.

Quick comparison

- `any`: no checking. Use rarely.
- `unknown`: any value, but you must narrow before use. Default for untrusted data.
- `never`: no value can exist here. Use for impossible states and exhaustive switches.

When you need reusable logic without `any`, generics are the better tool.

Handle null and undefined deliberately

Include absent values in a type and check them before using the value under strict null checking.

Missing values cause a large share of JavaScript crashes: `Cannot read properties of undefined`. TypeScript's answer is to make absence part of the type. A value that may be missing should say so:

```
function uppercase(value: string | undefined) {
  return value === undefined ? '' : value.toUpperCase()
}
```

The explicit check narrows `value` to `string` in the second branch. Skip the check and call `value.toUpperCase()` directly, and under strict null checking the compiler stops you:

```
'value' is possibly 'undefined'.
```

That error is the whole feature. The crash that would have happened at runtime, on some unlucky input, is now a compile-time message pointing at the exact expression.

Two kinds of absence

JavaScript has both `null` and `undefined`, and TypeScript keeps them distinct. Use them according to the real boundary. Missing object properties and functions without a return commonly produce `undefined`. An API might use `null` deliberately — JSON has `null` but no `undefined`, so parsed responses carry `null`. Match your types to what the data actually contains.

Do not silence the check

Do not hide absence with a non-null assertion:

```
value!.toUpperCase()
```

The `!` adds no runtime check. It only tells the compiler "trust me, this is not missing". If the value is missing, JavaScript still fails — with the exact crash strict null checking exists to prevent. Every `!` in a codebase is a place where the type system was overruled by hope.

Defaults with ??

Sometimes a default is the right behavior:

```
const label = value ?? 'Unknown'
```

The nullish coalescing operator uses the default only for `null` or `undefined`, not for valid empty strings. That distinction matters: `value || 'Unknown'` would also replace `''`, `0`, and `false`, which are often legitimate values. Reach for `??` when only genuine absence should trigger the fallback.

After either an explicit check or `??`, TypeScript narrows the type, and the rest of the code works with a plain `string`.

Exercise: write a function accepting `number | null`. Preserve `0` as valid input while returning a default only for `null`.

Check your understanding: types and inference

Check annotations, inference, primitives, arrays, tuples, any, unknown, void, and never.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Objects and unions

Describe object shapes and compose types deliberately.

Describe an object shape

Name the properties an object must have and let TypeScript check object literals and property access.

Most of the values your programs pass around are objects. An object type describes the properties your code expects:

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

const user: User = { id: 1, name: 'Ada' }
```

Once the shape has a name, TypeScript checks every use of it. It rejects a missing `name`, a string `id`, or an extra property on a directly checked object literal.

Leave out `name` and you get:

```
Property 'name' is missing in type '{ id: number; }' but required in type 'User'.
```

Add a property the type does not declare and you get:

```
Object literal may only specify known properties, and 'email' does not exist in type 'User'.
```

That second error only fires on object literals assigned directly to the type. It catches typos like `nmae` at the moment you write the object, which is exactly when they are cheapest to fix.

Structural typing

The check is structural. A value does not need to be created by a `User` constructor. It only needs compatible properties. If a function returns `{ id: 7, name: 'Grace' }`, that value is a valid `User`, no matter where it came from.

This makes ordinary JavaScript objects easy to type:

```
function printUser(user: User) {
```

```
  console.log(`${user.id}: ${user.name}`)
}
```

Inside the body, `user.id` is a `number` and `user.name` is a `string`. A typo like `user.fullName` fails immediately instead of printing `undefined` at runtime.

What the type does not do

The function promises to use the `User` shape. It does not validate unknown runtime input. If the object came from `JSON.parse()` or a network response, the annotation is a claim, not a check. Later in the course we validate uncertain data before trusting it.

My advice is to keep object types focused on the contract you need. A large type containing every possible API field makes functions harder to reuse. If `printUser()` only reads `id` and `name`, a two-property type is the honest contract.

Exercise: add an `active: boolean` property. Fix every error without using an assertion.

TypeScript interfaces vs types: which one to use

TypeScript interfaces vs type aliases: practical differences, unions, declaration merging, and which one to pick for a consistent codebase.

Pick one and stay consistent. I use **type aliases** for almost everything. They cover object shapes, unions, and primitives in one syntax.

Both `type` and `interface` can describe objects. You give a name to a shape, then use it on variables and function parameters.

Describing objects

With a type alias:

```
type Dog = {
  name: string
  age: number
}
```

With an interface:

```
interface Dog {
  name: string
  age: number
}
```

You use them the same way:

```
const jack: Dog = {
  name: 'Jack',
  age: 3
}
```

Optional properties work with both. Add `?` after the property name:

```
type Dog = {
  name: string
```

```
  age?: number
}
```

Interfaces can **extend** other interfaces:

```
interface Animal {
  name: string
}

interface Dog extends Animal {
  age: number
}
```

Type aliases use intersections for the same idea:

```
type Animal = {
  name: string
}

type Dog = Animal & {
  age: number
}
```

What only types can do

Type aliases can name **primitives**, **unions**, and **intersections**. Interfaces cannot.

```
type ID = string | number

type Status = 'idle' | 'loading' | 'done'

type Point = { x: number } & { y: number }
```

This is why I reach for `type` first. One keyword covers object shapes and these other patterns.

What only interfaces can do

Interfaces support **declaration merging**. If you declare the same interface twice, TypeScript merges them:

```
interface User {
  name: string
}

interface User {
  age: number
}

const jack: User = {
  name: 'Jack',
  age: 3
}
```

This is useful when a library adds types to a global object over multiple files. For your own app

code, merging is usually a footgun.

My verdict

For day-to-day app code, pick `type` or `interface` and stick with it. I prefer `type` because it handles unions and intersections without switching syntax.

If you work on a library that augments global types, interfaces have a clear edge. Otherwise the choice matters less than consistency across your team.

New to the language? Start with our TypeScript hub. Once you know the basics, make sure your `tsconfig.json` has `strict` enabled.

Optional and readonly properties

Express fields that may be absent and fields that should not be reassigned through a typed reference.

Object types support two useful modifiers. A question mark means a property may be absent. `readonly` prevents assignment through that typed reference:

```
type User = {
  readonly id: number
  nickname?: string
}
```

Both `{ id: 1 }` and `{ id: 1, nickname: 'ada' }` are valid `User` values. The type tells every reader of the code, and the compiler, that `nickname` is not guaranteed.

Reading an optional property

Reading `nickname` produces `string | undefined`, so narrow it before using string methods:

```
function label(user: User) {
  return user.nickname?.toUpperCase() ?? `User ${user.id}`
}
```

The optional chain calls `toUpperCase()` only when `nickname` exists. When it is absent, the expression evaluates to `undefined`, and `??` supplies the fallback. Call `user.nickname.toUpperCase()` without the `?.` and the compiler answers:

```
'user.nickname' is possibly 'undefined'.
```

Do not mark a property optional only because creating it is inconvenient. Optional means callers and runtime code must handle its absence. Every read pays that cost with a check. If the value always exists after construction, make it required and fix the construction site instead.

What readonly protects

`readonly` stops this assignment during checking:

```
user.id = 2
```

The error is direct:

```
Cannot assign to 'id' because it is a read-only property.
```

That is exactly what you want for identifiers: an `id` should be set once when the record is created and never reassigned.

Know the limits, though. `readonly` does not freeze the object at runtime. JavaScript can still mutate the same object through another reference that allows writing — an alias typed as `{ id: number }`, for example, writes freely. The modifier is a compile-time contract on one view of the object, not runtime protection like `Object.freeze()`.

In practice that contract is still worth a lot. Most accidental mutations happen through the typed references your own code holds, and `readonly` catches those.

Exercise: add an optional `bio` and render it without using `!` or `as string`.

TypeScript type for a string or array of strings

Learn how to declare a TypeScript type that can be either a string or an array of strings using a union, written as string or string[].

Simple one, but I forgot the exact syntax and had to search for it, so let me write it:

```
export interface Props {
  tag: string | string[]
}
```

`tag` can be a string, or an array of strings.

The `|` creates a **union type**: the property accepts either form. The same works with a type alias:

```
type Props = {
  tag: string | string[]
}
```

I hit this writing an Astro component that takes either one tag or a list of tags. Both call sites should work, and with the union they do:

```
const one: Props = { tag: 'typescript' }
const many: Props = { tag: ['typescript', 'javascript'] }
```

Reading the value back

The union is convenient for callers, but it puts a small burden on the reading side. When you read `tag`, TypeScript only allows operations that exist on both types. Call an array method directly and it complains:

```
Property 'map' does not exist on type 'string | string[]'.
  Property 'map' does not exist on type 'string'.
```

The fix is to narrow with `Array.isArray()`, which is the runtime check TypeScript understands for arrays:

```
function normalizeTags(tag: string | string[]) {
  return Array.isArray(tag) ? tag : [tag]
}
```

When the check is true, `tag` is a `string[]` and is returned as-is. When it is false, `tag` must be a `string`, so wrapping it in brackets produces an array. The return type is inferred as `string[]` without any annotation.

This is the pattern I recommend: accept two convenient input shapes at the boundary, then normalize them immediately. Call `normalizeTags()` once, and the rest of the component works with

a plain `string[]` — no repeated `Array.isArray()` checks scattered through the code.

The same union works anywhere, not just in component props. A function parameter typed `string | string[]` gives callers the same flexibility, with the same one-line normalization inside.

Exercise: reject an array containing a number, then hover over the result of `normalizeTags('typescript')`.

Restrict values with literal unions

Model a small closed set of allowed strings or numbers instead of accepting every value of the primitive type.

A **literal type** is a type with exactly one value. `'loading'` is a type whose only member is the string `'loading'`. On its own that is not very useful. Combined in a union, it becomes one of the most practical tools in TypeScript.

A literal union makes invalid states harder to represent:

```
type Status = 'idle' | 'loading' | 'success' | 'error'
```

Use it at the boundary:

```
function setStatus(status: Status) {}

setStatus('loading')
setStatus('loadng')
```

The misspelled value fails during checking:

```
Argument of type '"loadng"' is not assignable to parameter of type 'Status'.
```

Callers also get autocomplete for the allowed states. Type the opening quote and the editor lists all four values.

A plain `string` would accept every spelling, even though the application understands only four values. The union makes that closed set visible, and it turns "which statuses exist?" from a documentation question into something the compiler answers.

Watch for widening

Literal types can widen when values move through mutable objects:

```
const request = { status: 'loading' }
```

Because `request.status` can be reassigned, TypeScript usually infers `string`. Pass `request.status` to `setStatus()` and you get:

```
Argument of type 'string' is not assignable to parameter of type 'Status'.
```

Annotate the object or use an appropriate const assertion when you need the literal preserved:

```
const request: { status: Status } = { status: 'loading' }
const fixed = { status: 'loading' } as const
```

The annotation keeps the property assignable to other `Status` values. The `as const` version locks it to exactly `'loading'` and makes it readonly. Choose based on whether the object should change.

The runtime is still plain strings

The runtime values remain ordinary strings. Nothing at runtime prevents a network response from carrying `'cancelled'` or garbage. Validate outside data before treating it as `Status`; the union is a compile-time contract between parts of your own code.

Exercise: add a `'cancelled'` state and follow the compiler errors to every place that needs a decision.

Combine requirements with intersections

Create a type that must satisfy two object shapes and recognize when one explicit object type is easier to read.

A union says a value is one type *or* another. An **intersection** says a value is one type *and* another. It combines requirements: a value must satisfy every member.

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

type SavedUser = User & Timestamped
```

A `SavedUser` needs the properties from `User` and `Timestamped`:

```
const user: SavedUser = {
  id: 1,
  name: 'Ada',
  createdAt: new Date()
}
```

Leave out `createdAt` and the assignment fails: the value satisfies `User` but not `Timestamped`, so it is not the intersection.

Intersections are useful when independent concerns truly meet in one value. `Timestamped` can decorate users, posts, and comments without repeating the field in each type. A function that only cares about timestamps can accept `Timestamped` alone, and any `SavedUser` qualifies through structural typing.

Conflicting properties

Be careful with conflicting properties:

```
type A = { id: string }
type B = { id: number }
type Impossible = A & B
```

The `id` would need to be both a string and a number, so no ordinary value can satisfy it. TypeScript resolves the property to `never`, and every assignment of a real `id` fails:

```
Type 'string' is not assignable to type 'never'.
```

The compiler does not reject the `Impossible` type itself, only every attempt to build a value for it. If you see `never` appearing in an error about a property you defined twice, an intersection conflict is the usual cause.

When to prefer one explicit type

Deeply layered intersections also produce difficult error messages. A missing property in `A & B & C & D` gets reported against the whole chain, and you have to work out which member wanted it.

My advice: if the combined shape is a core domain object that you name, read, and discuss often, one explicit object type can be easier to read. Keep intersections for genuinely independent, reusable capabilities.

Exercise: combine `{ title: string }` with `{ updatedAt: Date }`, then intentionally omit one property and read the error.

Check your understanding: objects and unions

Check type aliases, interfaces, optional and readonly properties, unions, intersections, and literal types.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Functions and generics

Type calls, callbacks, reusable functions, and constraints.

Type function parameters

Put type annotations on a function boundary so invalid calls fail before the function body executes.

Function parameters are boundaries. Inside the body you control the code; at the boundary, anyone can call with anything. Annotate what callers are allowed to pass:

```
function greet(name: string, times = 1) {
  return name.repeat(times)
}
```

Two things happen here. `name: string` is an explicit annotation. The default value `= 1` lets TypeScript infer `times` as a number without one — and it also makes the parameter optional, since omitting it uses the default.

These calls are valid:

```
greet('Ada')
greet('Ada', 3)
```

A boolean name or string count fails before the function runs:

```
Argument of type 'boolean' is not assignable to parameter of type 'string'.
```

Compare that with plain JavaScript, where `greet(true, 3)` would reach `.repeat()` and throw at runtime, possibly in production. The annotation moves the failure to the call site, at compile time, with the caller's file and line in the message.

Unlike variables, parameters usually need explicit annotations. There is no initializer to infer from, and under strict checking an unannotated parameter is an error: `Parameter 'name' implicitly has an 'any' type.`

Optional parameters

Mark a parameter with `?` only when callers can genuinely omit it:

```
function greet(name: string, title?: string) {
  return title ? `${title} ${name}` : name
}
```

Inside the function, `title` is `string | undefined`. The implementation must handle both cases — that is the deal you make when you accept an optional argument. Here the truthiness check handles it, and `greet('Lovelace', 'Countess')` returns `'Countess Lovelace'`.

Order matters too: optional parameters must come after required ones, because a call like `greet(undefined, 'Ada')` has no way to skip a leading parameter cleanly.

Do not make things optional to silence errors

Do not make required data optional just to silence call-site errors. That moves the same problem into every use inside the function: each read of the parameter now needs an `undefined` check. If a caller does not have the data, the fix is at that caller, not in the signature everyone else shares.

Exercise: add an optional punctuation parameter with a useful default. Inspect its inferred type inside the function.

Use inferred and explicit return types

Let simple implementation details infer their result and annotate public boundaries when the promised return type matters.

TypeScript infers a return type from every return path. You rarely have to write one, and for small local functions inference is often enough.

For example, this result is inferred as `number`:

```
function total(values: number[]) {
  return values.reduce((sum, value) => sum + value, 0)
}
```

Hover over `total` in your editor and you see `(values: number[]) => number`. The compiler worked that out from the `reduce()` call and its `0` seed.

When to annotate

An explicit return annotation is useful when the function promises a public contract:

```
function total(values: number[]): number {
  return values.reduce((sum, value) => sum + value, 0)
}
```

The difference shows when the implementation changes. Suppose someone edits the function to return `values.length ? sum : 'none'`. Without the annotation, the inferred type quietly becomes `string | number`, and the errors appear in every caller that does arithmetic with the result. With `: number`, the error appears inside the function, at the bad `return` statement:

```
Type 'string' is not assignable to type 'number'.
```

Now an accidental string return fails inside the function instead of changing the type seen by every

caller. The person who made the change sees the error, in the file they were editing. That locality is the main argument for annotating exported functions.

Annotations also help recursive functions and functions with several branches. A recursive function sometimes cannot infer its own result. Multiple branches make it easy to miss a path that returns `undefined`. Writing the return type makes you decide whether every path returns a value.

When not to annotate

Do not annotate every small local function. Inference keeps implementation code readable and often preserves a more precise type — a function returning `'asc'` infers the literal type `'asc'`, while a hasty `: string` annotation would throw that precision away.

My rule: annotate the exported surface of a module, let the internals infer.

Exercise: add an empty-array branch returning `'none'`. Compare the inferred return type with the error produced by the explicit `: number` contract.

Type callbacks

Describe the parameters and result of a function passed as a value.

In JavaScript, functions are values. You pass them to array methods, event handlers, and utilities all the time. TypeScript needs a way to describe those values, and that is what a **function type** does.

A function type describes how another function may be called:

```
function transform(value: string, fn: (value: string) => string) {
  return fn(value)
}
```

Read `(value: string) => string` as "a function that takes one string and returns a string". The arrow separates parameters from the result. Now `transform()` can only receive callbacks matching that contract — passing `(n: number) => n * 2` fails at the call site, not deep inside the body.

Parameter names inside the type are documentation; compatibility depends on their types and positions. `(text: string) => string` and `(value: string) => string` are the same type.

Inference inside inline callbacks

Context gives an inline callback its parameter type:

```
const result = transform('hello', value => value.toUpperCase())
```

You do not need to annotate `value` again. TypeScript looks at the type of `fn` and knows `value` is a `string`. This is why callbacks in `.map()` and `.filter()` rarely need annotations: the array's element type flows in.

void callbacks

Use `void` when the caller ignores the callback result:

```
function visit(fn: (title: string) => void) {
  fn('TypeScript')
}
```

A `void` return type means "whatever this returns, I will not use it". You can still pass a callback that returns something — `visit(title => title.length)` is fine — because ignoring a value is always safe. That flexibility is deliberate and makes `void` the right default for handlers and listeners.

One trap with optional parameters

Be careful with optional callback parameters. Writing `(index?: number) => void` means your code may call the callback without an index. It does not mean callers can ignore a required argument you always provide. If you always pass the index, declare it required: callers who do not need it can write a callback with fewer parameters, which TypeScript accepts.

Exercise: change `transform()` so the callback converts a string into a number. Confirm that the function result becomes `number`.

Object destructuring with types in TypeScript

Learn the correct TypeScript syntax for adding types to object destructuring, why name: string fails, and how a dedicated type or interface keeps it clean.

I was using TypeScript in Deno to build a sample project and I had to destructure an object. I am familiar with TypeScript basics but sometimes I hit a problem.

Object destructuring was one of those.

I wanted to do

```
const { name, age } = body.value
```

I tried adding the `string` and `number` types like this:

```
const { name: string, age: number } = body.value
```

But this didn't work. It apparently worked, but in reality this is assigning the `name` property to the `string` variable, and the `age` property value to the `number` variable.

That is because a colon inside a destructuring pattern already has a meaning in JavaScript: renaming. `{ name: string }` means "take the `name` property and put it in a variable called `string`". No error, no types, just two badly named variables. That is why the mistake is sneaky — the code runs.

The correct syntax is this:

```
const { name, age }: { name: string; age: number } = body.value
```

The annotation goes after the whole pattern, and it types the object being destructured, not the individual variables. TypeScript then works out that `name` is a `string` and `age` is a `number` from the object type.

The best way to approach this would be to create a type or interface for that data:

```
interface Dog {
  name: string
  age: number
}
```

Then you can write the above in this way, which is shorter:

```
const dog: Dog = body.value
```

And once the value is typed, destructuring needs no annotation at all:

```
const { name, age } = dog
```

TypeScript already knows the shape of `dog`, so `name` and `age` get their types by inference. This is the version I end up with most of the time: type the value once, destructure freely afterwards.

The same pattern-then-annotation syntax works in function parameters, where destructuring is common:

```
function describeDog({ name, age }: Dog) {
  return `${name} is ${age} years old`
}
```

Again the type applies to the whole parameter object, and the destructured variables are inferred from it.

TypeScript generics explained simply

TypeScript generics let functions and types work with many values while keeping type safety. Learn syntax, constraints, and when to skip them.

Generics let you write code that works with many types without throwing away type information. You get one function or type definition, and TypeScript still knows what you passed in.

Why generics exist

Say you want a function that returns the first item of an array.

Without generics you pick one type:

```
function firstNumber(items: number[]) {
  return items[0]
}
```

That works for numbers. For strings you copy the function and change the type. That gets old fast.

You could use `any`:

```
function first(items: any[]) {
  return items[0]
}
```

TypeScript stops helping you. You lose autocomplete and compile-time checks.

Generics fix this. You write the function once, and the type follows the argument:

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

const n = first([1, 2, 3])   // number | undefined
const s = first(['a', 'b'])  // string | undefined
```

The `T` is a **type parameter**. TypeScript fills it in when you call the function.

Generic constraints with extends

Sometimes the generic type must have certain properties.

You restrict it with `extends`:

```
interface HasName {
  name: string
}

function greet<T extends HasName>(person: T) {
  console.log(`Hi ${person.name}!`)
}

greet({ name: 'Flavio', age: 40 })
```

This is the same idea from the TypeScript tutorial: generics can be limited to a class family or interface.

Generic types and interfaces

Generics are not just for functions. You can type API responses like this:

```
type ApiResponse<T> = {
  data: T
  status: number
}

type User = { id: number, name: string }

const res: ApiResponse<User> = {
  data: { id: 1, name: 'Flavio' },
  status: 200
}
```

You write `ApiResponse` once. Swap `User` for any other type.

Arrays use the same pattern:

```
const nums: Array<number> = [1, 2, 3]
```

`Array<number>` is a generic type. The number inside the angle brackets is the type argument.

When not to use generics

My advice is to skip generics when a plain type works.

If your function only ever handles strings, type it as `string[]`. No generic needed.

If you reach for generics on the first version of a helper, you might be over-engineering. Start simple. Add a generic when you actually need the same logic for a second type.

Generics shine in reusable utilities, API wrappers, and libraries. For a one-off function in your app, a concrete type is often enough.

Constrain a generic only when necessary

Require a capability from a type parameter while preserving the caller-specific type in the result.

A generic preserves a relationship between caller-specific types. A constraint adds the minimum capability the implementation needs:

```
function longest<T extends { length: number }>(a: T, b: T): T {
  return a.length >= b.length ? a : b
}
```

Strings and arrays both work because they have a numeric `length`. A number fails because it does not.

The return type remains `T`, not just `{ length: number }`. Passing two string arrays produces a string-array result.

A constraint is not permission to create any matching object and return it as `T`:

```
function broken<T extends { length: number }>(value: T): T {
  return { length: value.length }
}
```

The object satisfies the constraint, but it may lack caller-specific properties required by `T`. TypeScript correctly rejects it.

Use a generic when a type parameter relates two or more positions. If it appears only once, a normal parameter type is usually clearer.

Exercise: call `longest()` with two objects that also have a `label`. Confirm that the returned value keeps the label type.

Check your understanding: functions and generics

Check parameter and return types, optional values, callbacks, destructuring, generic inference, and constraints.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

Narrowing uncertain values

Refine unions with checks, guards, and discriminants.

Narrow with typeof

Refine a primitive union by checking the runtime kind before using type-specific operations.

A union describes every allowed value at the boundary. A runtime check tells TypeScript which member exists on the current path:

```
function format(value: string | number) {
  return typeof value === 'number' ? value.toFixed(2) : value.trim()
}
```

Inside the first branch, `value` is a number. In the other branch, it must be a string. This is **narrowing**.

The check exists at runtime, and TypeScript follows its control flow. If you move the number operation before the check, the error returns.

Remember one JavaScript edge case: `typeof null` is `'object'`. A check for an object usually also needs `value !== null`.

The union remains honest at the boundary, while each branch gets the operations it can safely use.

Exercise: extend the function to accept `boolean`. Add a branch that returns `'yes'` or `'no'`, and hover over `value` in every branch.

Narrow with equality, null checks, and truthiness

Choose a check that preserves valid values instead of accidentally discarding empty strings or zero.

An `if (value)` check narrows a type, but it narrows more than most people intend. Truthiness removes every falsy possibility, not only missing values.

```
function printCount(count: number | undefined) {
  if (count) {
    console.log(count)
  }
}
```

Inside the `if`, TypeScript narrows `count` to `number`, and the code compiles. But this branch skips `undefined`, and it also skips the valid number `0`. Call `printCount(0)` and nothing prints. The compiler cannot flag this, because the narrowing is technically correct — the bug is in what the check throws away.

The same trap hits strings, where the empty string `''` is falsy, and booleans, where `false` is falsy. A form field left blank and a form field never submitted are different situations, and truthiness collapses them into one.

Prefer explicit comparisons

Use an explicit check when zero, `false`, or an empty string has meaning:

```
if (count !== undefined) {
  console.log(count)
}
```

Now `printCount(0)` prints `0`, and TypeScript still narrows `count` to `number` inside the branch. The check states exactly what you are excluding, nothing more.

You can remove both `null` and `undefined` with `value != null`. The loose `!=` deliberately treats the two as equal, so one comparison covers both. TypeScript understands the idiom and narrows correctly. Use it intentionally and explain the uncommon loose-equality choice to your team, because most style guides ban `!=` everywhere else.

Equality narrows both sides

Equality can also relate two unions. If `a` is `string | number` and `b` is `string | boolean`, the branch `a === b` narrows both to `string`, their only shared possibility:

```
function match(a: string | number, b: string | boolean) {
  if (a === b) {
    a.toUpperCase()
    b.toUpperCase()
  }
}
```

For `a === b` to be true, both values must have the same type, and `string` is the only type in both unions. TypeScript follows that logic without any annotation.

My advice: reserve truthiness for values where all falsy inputs genuinely mean "nothing to do", and reach for `!== undefined` everywhere else.

Exercise: write a function accepting `string | undefined`. Preserve an empty string while replacing only `undefined` with `'Unknown'`.

Narrow with in and instanceof

Use property checks for object unions and constructor checks for real runtime classes.

`typeof` only distinguishes primitives. Every object answers `'object'`, so a union of two object shapes needs a different runtime question. You have two: does this property exist, and was this built by that constructor?

Narrow with in

The JavaScript `in` operator checks whether a property exists on an object or its prototype chain. TypeScript can use that fact to narrow an object union:

```
type Success = { data: string }
type Failure = { message: string }

function print(result: Success | Failure) {
  if ('message' in result) {
    console.error(result.message)
    return
  }

  console.log(result.data)
}
```

Inside the `if`, only `Failure` has a `message` property, so `result` is a `Failure`. After the early return, TypeScript knows the remaining code deals with `Success`, and `result.data` compiles.

Without the check, `result.data` fails:

```
Property 'data' does not exist on type 'Success | Failure'.
  Property 'data' does not exist on type 'Failure'.
```

One caveat: optional properties can appear in both branches because they may exist or be absent. If both variants declare `message?: string`, the `in` check proves nothing. A required discriminant is clearer when variants overlap.

Narrow with instanceof

`instanceof` checks the prototype chain:

```
function logWhen(value: Date | string) {
  if (value instanceof Date) {
    console.log(value.toISOString())
  } else {
    console.log(value)
  }
}
```

In the first branch `value` is a `Date`, so `toISOString()` is available. In the `else` branch it must be a `string`.

This works for runtime constructors such as `Date`, `Error`, `URL`, and your JavaScript classes. It cannot use a type alias or interface because those disappear during compilation. Writing `value instanceof Success` fails with:

```
'Success' only refers to a type, but is being used as a value here.
```

That error is the compiler reminding you which check you need. When the variants are plain object shapes, reach for `in` or a discriminant field. When they are class instances, `instanceof` is the direct, honest check.

Exercise: create a union of `Date | string`, narrow it with `instanceof`, and format both branches.

Model states with a discriminated union

Give each variant a shared literal field so a switch can safely expose the fields belonging to that state.

A **discriminated union** models a value that is always in exactly one of several states. Use one shared literal property to identify every valid variant:

```
type Result =
  | { status: 'success'; data: string }
  | { status: 'error'; message: string }
```

The `status` property is the discriminant. Every variant has it, and each variant gives it a different literal value. That single field is enough for TypeScript to tell the variants apart.

Try reading `result.data` before checking `status` and the compiler stops you:

```
Property 'data' does not exist on type 'Result'.
  Property 'data' does not exist on type '{ status: "error"; message: string; }'.
```

The error is precise: `data` might not exist, because the value could be the error variant.

Narrow with a switch

After checking the discriminant, TypeScript knows which other fields exist:

```
function printResult(result: Result) {
  switch (result.status) {
    case 'success':
      console.log(result.data)
```

```
      break
    case 'error':
      console.error(result.message)
      break
  }
}
```

Inside the `'success'` case, `result` is the success variant, so `result.data` is a plain `string`. Inside `'error'`, only `message` exists. No casts, no optional chaining, no extra checks.

Why not optional fields

This is safer than one object with optional `data` and `message` fields:

```
type WeakResult = {
  status: string
  data?: string
  message?: string
}
```

That weaker model allows impossible combinations, such as success without data, or an object carrying both fields at once. Every read becomes `string | undefined`, and you pay for the vagueness in every function that touches the value.

Discriminated unions work well for requests, reducers, messages, and UI states. Each state carries exactly the data it needs, and the compiler enforces the pairing.

One practical tip: when you add a variant later, search for every `switch` on the discriminant. TypeScript flags the branches where the new variant leaks through, which turns a risky change into a guided checklist.

Exercise: add a `{ status: 'loading' }` variant. Follow the compiler and update code that assumed only success or error.

Type predicates and assertions

Write a reusable guard when normal checks are not enough, and treat assertions as claims that require evidence.

Inline `typeof` and `in` checks narrow types where you write them. But when the same check appears in five places, you want to extract it into a function — and a plain function returning `boolean` loses the narrowing. A **type predicate** fixes that: it lets a reusable runtime check communicate its result to TypeScript.

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

function isUser(value: unknown): value is User {
  if (typeof value !== 'object' || value === null) return false
  if (!('id' in value) || !('name' in value)) return false

  return typeof value.id === 'number' && typeof value.name === 'string'
}
```

The return type `value is User` is the predicate. It replaces `boolean` and adds a promise: when

this function returns `true`, treat the argument as a `User`.

After `isUser(input)` returns true, TypeScript narrows `input` to `User`:

```
const input: unknown = JSON.parse('{"id": 7, "name": "Grace"}')

if (isUser(input)) {
  console.log(input.name.toUpperCase()) // GRACE
}
```

Without the guard, `input.name` fails, because `unknown` exposes nothing. Inside the branch, both properties are available with their exact types.

The predicate is trusted, not verified

Here is the part that deserves respect. The predicate is a promise made by your implementation. TypeScript does not verify that the boolean logic proves every property. Delete the `typeof value.name === 'string'` check and the compiler still believes the `value is User` claim — the guard just lies now.

That makes guards a small trusted core of your codebase. Test them with valid, missing, wrong-type, `null`, and array inputs. `null` and arrays matter specifically because `typeof null` and `typeof []` are both `'object'`, the classic way guards go wrong.

Assertions skip the check entirely

An assertion is different:

```
const user = input as User
```

`as User` performs no runtime validation and involves no check at all. You are overriding the compiler, and if you are wrong, the failure surfaces later as a confusing runtime error far from this line.

Use an assertion only when external evidence proves something the checker cannot see — a DOM element you just created, for example. Prefer normal narrowing or parsing for uncertain input. A predicate documents and executes its evidence; an assertion has none.

Exercise: remove the `name` check from `isUser()`. Notice that TypeScript still trusts the predicate, then explain why tests matter.

Type narrowing and type guards in TypeScript

Type narrowing lets TypeScript refine union types as you check them. Learn typeof, truthiness, in, instanceof, discriminated unions, and is guards.

Type narrowing is how TypeScript refines a union type as you check it. You start with a value that could be one of several types. After a check, TypeScript knows the exact type in that branch.

This builds on union types, where a value can be one type or another.

typeof

For primitives, `typeof` is the simplest guard:

```
function double(value: string | number) {
```

```
  if (typeof value === 'string') {
    return value.toUpperCase()
  }
  return value * 2
}
```

Inside the `if` block, `value` is a `string`. Outside, it is a `number`.

Truthiness

Checking for `null` or `undefined` narrows optional values:

```
function greet(name: string | undefined) {
  if (!name) {
    return 'Hello stranger'
  }
  return `Hello ${name}`
}
```

After the check, `name` is a `string`.

The in operator

For objects, `in` checks if a property exists:

```
type Dog = { bark: () => void }
type Cat = { meow: () => void }

function speak(pet: Dog | Cat) {
  if ('bark' in pet) {
    pet.bark()
  } else {
    pet.meow()
  }
}
```

instanceof

For class instances, use `instanceof`:

```
function logDate(value: Date | string) {
  if (value instanceof Date) {
    console.log(value.toISOString())
  } else {
    console.log(value)
  }
}
```

Discriminated unions

The pattern I use most is a shared `kind` field. Each variant in the union carries a literal type on that field.

```
type ApiResult =
```

```
  | { kind: 'ok', data: { name: string } }
  | { kind: 'error', message: string }

function handle(result: ApiResult) {
  switch (result.kind) {
    case 'ok':
      console.log(result.data.name)
      break
    case 'error':
      console.log(result.message)
      break
  }
}
```

TypeScript connects `kind: 'ok'` to the `data` property and `kind: 'error'` to `message`. No extra checks needed inside each branch.

This pairs well with `never` for exhaustive switches, which we cover in any vs unknown vs never.

Custom type guards with is

You can write a function that tells TypeScript what type a value is:

```
type Fish = { swim: () => void }
type Bird = { fly: () => void }

function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined
}

function move(pet: Fish | Bird) {
  if (isFish(pet)) {
    pet.swim()
  } else {
    pet.fly()
  }
}
```

The `pet is Fish` return type is the type guard. When the function returns `true`, TypeScript narrows `pet` to `Fish`.

Custom guards help when the check is too complex for a one-line `typeof` or `in` test.

If you validate data at runtime too, libraries like Zod narrow types after parsing. For compile-time unions you control, discriminated unions and `is` guards are the tools you reach for first.

Check your understanding: narrowing

Check typeof, truthiness, property and constructor guards, discriminated unions, predicates, assertions, and exhaustive branches.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The compiler and runtime

Configure strict checking and understand where types stop.

Create a tsconfig.json

Define the project boundary and compiler behavior once instead of passing unrelated flags on every command.

Without a configuration file, every `tsc` invocation needs the same flags repeated on the command line, and your editor has no way to know which rules apply. A `tsconfig.json` fixes both problems: it marks a TypeScript project root and records the compiler settings once.

Create a starting configuration:

```
npx tsc --init
```

This generates a `tsconfig.json` full of commented-out options. It works, but I prefer starting from a small explicit file instead:

```
{
  "compilerOptions": {
    "strict": true,
    "target": "ES2022",
    "module": "NodeNext",
    "moduleResolution": "NodeNext"
  },
  "include": ["src/**/*.ts"]
}
```

Each part has a job. The `include` and `exclude` patterns choose the source files, while `compilerOptions` controls checking and output. `target` decides which JavaScript syntax the compiler may emit. `module` and `moduleResolution` decide how imports are written and resolved.

The correct module settings depend on the runtime and build tool. `NodeNext` is right for modern Node.js. A browser bundle built with Vite typically wants `"module": "ESNext"` with `"moduleResolution": "bundler"`. Do not copy them blindly between a Node.js app and a browser bundle.

Verify the project boundary

Run the project compiler without file arguments:

```
npx tsc --noEmit
```

When `tsc` runs with no file arguments, it walks up from the current directory, finds `tsconfig.json`, and checks exactly the files matched by `include`. No output means every file passed.

Passing individual files can bypass the project configuration you intended to test. `npx tsc src/index.ts` ignores your `tsconfig.json` entirely and falls back to default compiler options, so it can pass while the real project check fails.

A common failure mode

If the checker seems to ignore a file, it is usually outside the `include` patterns. Confirm what the project actually contains:

```
npx tsc --noEmit --listFiles
```

The output lists every file in the compilation, including library declaration files. If your file is missing from the list, fix the glob rather than the file.

Keep `tsconfig.json` in version control so editors, local checks, and continuous integration agree.

Exercise: create one `.ts` file inside `src` and another outside it. Run `npx tsc --noEmit --listFiles` and inspect which files belong to the project.

Enable strict checking

Turn on the family of strict compiler checks and fix the contracts they reveal instead of weakening the project globally.

Start with:

```
{
  "compilerOptions": {
    "strict": true
  }
}
```

`strict` enables a family of stronger checks. These include implicit-`any`, null-related, function, property-initialization, and other checks.

For example, strict checking rejects an untyped parameter:

```
function uppercase(value) {
  return value.toUpperCase()
}
```

It also rejects a possibly missing value before property access.

Fix the contract instead of turning strictness off globally. Add the real parameter type, narrow nullable values, and validate uncertain data.

When migrating an existing project, enable strict checks in a controlled sequence if necessary. Keep a list of temporary exceptions and remove them. New projects should begin strict.

TypeScript can add stricter behavior under this family in future versions. Read upgrade notes and run the checker before updating the compiler in production code.

Exercise: enable `strict`, add one untyped parameter and one possibly undefined value, then fix both without `any`, `!`, or assertions.

Type-check without emitting files

Use the compiler as a checker when another tool already transforms and bundles the source.

In most modern projects, `tsc` never produces the JavaScript you ship. Vite, esbuild, or another build tool strips the types and bundles the output, because doing that without type checking is dramatically faster. That leaves `tsc` with one job: checking.

Run the project checker without writing output:

```
npx tsc --noEmit
```

The compiler checks the complete project but does not write JavaScript, source maps, or declarations. Silence means success. Errors come out in the usual file, line, and message format, and the exit code is non-zero, which is what scripts and CI care about.

Why the bundler is not enough

The two tools answer different questions. A bundler asks whether it can transform and package the source. TypeScript asks whether the program satisfies its type contracts.

A fast transpiler can produce JavaScript even when the types are wrong. This code bundles without complaint:

```
const port: number = 'three thousand'
```

It is valid JavaScript once the annotation is stripped. Only the type checker reports the problem:

```
Type 'string' is not assignable to type 'number'.
```

So a green build proves very little about your types. Teams learn this the hard way when a refactor breaks twenty call sites and the deploy succeeds anyway.

Make checking a named command

Keep a separate type-check command:

```
{
  "scripts": {
    "typecheck": "tsc --noEmit"
  }
}
```

Run it locally and in continuous integration. A successful bundle does not replace it. In CI, run `npm run typecheck` as its own step next to tests, so a type failure is reported as exactly that instead of hiding inside a build log.

You can also set `"noEmit": true` in `tsconfig.json` `compilerOptions` instead of passing the flag, which keeps the script shorter and makes the project's intent explicit: this configuration exists for checking, another tool owns the output.

One habit worth building: run the typecheck before you commit, not just before you deploy. The errors are cheapest when the code is still fresh in your head.

Exercise: introduce a type error that still uses valid JavaScript syntax. Confirm that your build tool can transform it while `npm run typecheck` rejects it.

Understand type erasure

Know which TypeScript features disappear and why a type cannot be inspected like a JavaScript value.

TypeScript removes type-only syntax when it emits JavaScript.

```
type User = { name: string }
```

```
function greet(user: User) {
  return `Hello ${user.name}`
}
```

Compile that and look at the output:

```
function greet(user) {
  return `Hello ${user.name}`
}
```

The emitted JavaScript has the function, but no `User` alias or parameter annotation. The alias did not become some runtime registry entry. It is gone.

This is **type erasure**. Interfaces, aliases, annotations, and type-only imports exist for checking and editor tools. JavaScript cannot inspect them later, because by the time the code runs, they no longer exist anywhere.

Types are not values

That is why this cannot work:

```
value instanceof User
```

The compiler rejects it with a very literal message:

```
'User' only refers to a type, but is being used as a value here.
```

`User` is not a runtime value. `instanceof` is a JavaScript operator that needs a constructor function on its right side, and after erasure there is nothing there. The same applies to ideas like `typeof value === User` or looping over a type's properties — the runtime has no types to consult.

When you need a runtime decision, use something that survives compilation. Use a real class with `instanceof`, check object properties, or parse the value with a schema. A class works because a class declaration is both a type and a value: it emits a constructor function that exists at runtime.

What erasure buys and costs

Erasure also means types do not add runtime cost or automatically validate data. The emitted program remains ordinary JavaScript, exactly as fast as the untyped version, with no library shipped to your users.

The cost is the flip side: annotating a network response as `User` changes nothing about the actual bytes that arrive. The annotation is checked against your code, never against the data. Keeping "compile-time claim" and "runtime fact" separate in your head is one of the most useful mental models in this whole course.

Exercise: compile the example and inspect the output. List every TypeScript-only part that disappeared.

Use declaration files

Understand how .d.ts files describe JavaScript APIs and where library types come from.

A declaration file contains type information without runtime implementation.

For example, a JavaScript package might export `slugify()`. A declaration can describe it:

```
declare function slugify(value: string): string
```

```
export { slugify }
```

The `.d.ts` file tells TypeScript how callers may use the JavaScript. It does not provide the function at runtime.

Many packages ship their own declarations. Inspect the package metadata or editor types before installing anything else.

For older JavaScript libraries, a matching package under `@types` may provide community-maintained declarations:

```
npm install --save-dev @types/express
```

Do not install `@types` packages automatically when the library already includes types.

If a declaration disagrees with the real JavaScript, the checker can approve code that fails at runtime. Treat declarations as contracts that must match the implementation.

Exercise: write a declaration claiming a function returns `number`, then make the JavaScript return a string. Notice which side TypeScript trusts.

Validate data at runtime

Treat JSON, environment variables, form fields, and network responses as unknown until code has checked their actual shape.

Types are erased before your program runs, so nothing checks the data that arrives from outside. `JSON.parse()` returns whatever the text contained, no matter what type you wish it had. The honest move is to start uncertain input as `unknown`. This forces you to prove a useful shape before accessing fields.

Write the proof as a type predicate:

```
type Task = { id: number; title: string }

function isTask(value: unknown): value is Task {
  if (typeof value !== 'object' || value === null) return false
  if (!('id' in value) || !('title' in value)) return false

  return typeof value.id === 'number' && typeof value.title === 'string'
}
```

The checks build on each other. First rule out non-objects and `null` — remember `typeof null` is `'object'`, so the explicit `null` check is not optional. Then confirm the properties exist. Only then is it safe to inspect their types.

Use the guard at the boundary:

```
const input: unknown = JSON.parse(text)

if (!isTask(input)) {
  throw new Error('Invalid task data')
}

console.log(input.title)
```

After the guard, `input` is a `Task`, and the property access compiles. Before it, `input.title` fails with `'input' is of type 'unknown'.` — the compiler holding the line until you validate.

Test it against hostile input: `isTask(null)`, `isTask([])`, and `isTask({ id: '7', title: 'Ship' })` all return `false`. The string `'7'` is the case annotations alone would never catch.

The runtime check protects the running program. The type predicate describes what is true after validation. Neither one replaces the other.

Where to validate

Keep validation close to JSON, network, form, storage, and environment boundaries. Validate once, as the data enters, then pass the trusted `Task` inward. The rest of the application can then work with trusted values and never repeat the checks.

Hand-written guards scale poorly past a few fields. For larger contracts, a schema library can produce consistent errors and inferred types from one definition. Still test missing fields, wrong types, `null`, arrays, and oversized values — the library runs your schema, and the schema can be wrong.

Exercise: add a boolean `completed` field and verify that the guard rejects the string `'false'`.

Build a typed task list

Combine object types, literal unions, functions, narrowing, generics, and runtime parsing in one small program.

Build a command-line task list that reads and writes a JSON file.

Start with a state model that prevents impossible combinations:

```
type TaskStatus = 'open' | 'completed'

type Task = {
  id: number
  title: string
  status: TaskStatus
}
```

Support four commands: add, list, complete, and remove. Give every function a clear input contract. Let local return types infer unless an exported boundary needs an explicit promise.

Treat the JSON file as `unknown`. Validate the top-level array and every task field before returning `Task[]`. A type assertion does not count as parsing.

Add this helper only if it preserves useful information:

```
function findById<T extends { id: number }>(items: T[], id: number) {
  return items.find(item => item.id === id)
}
```

The result keeps the caller's specific item type and includes `undefined` when no item matches.

Test these failure paths:

- the file does not exist
- the JSON is malformed

- one task has a string ID
- an unknown command is passed
- a requested task does not exist

Keep `strict` enabled and add a repeatable check:

```
npx tsc --noEmit
```

Finish by removing any `any`, unjustified assertion, or non-null assertion. The goal is not to make the checker quiet. The goal is to make every uncertain boundary and missing value explicit.

Check your understanding: compiler and runtime

Check tsconfig, strict mode, erased types, declarations, runtime validation, noEmit, and ecosystem basics.

Answer each question before continuing. Read the explanation after every answer, including the ones you get right.

Take this interactive quiz online

The interactive version of this course is available at https://flaviocopes.com/courses/typescript/.