



TypeScript

Version 5.9 · verified 2026-07-25

A gradual, compact walk through TypeScript's type system — basics to advanced.

Contents

Mental model	2
Annotating & inferring types	2
Object shapes: type vs interface	3
Unions, literals & narrowing	3
Functions	5
Generics	5
Utility types	6
Literal types, <code>as const</code> & <code>satisfies</code>	7
Advanced type-level programming	7
Modules & strict config essentials	8
Further reading	9

Mental model

Types are erased at compile time — nothing you write in a type annotation survives to runtime, which is why validating a value at a boundary needs a real check, not a type. Compatibility is structural, not nominal: two unrelated shapes with the same members are interchangeable, which explains most “why did that assignment work” surprises.

Annotating & inferring types

Syntax	Meaning
<code>let x: string</code>	Explicit annotation
<code>let x = "hi"</code>	Inferred as <code>string</code>
<code>unknown</code> vs <code>any</code>	Safe unknown vs checking off

Inference is the default and usually better than an annotation. Annotate boundaries — parameters, exported return types — and let inference handle the rest.

```
let a = "hi";           // inferred: string
let b: string = "hi";  // same type, redundant
let c: unknown = JSON.parse("{}");
```

Tip: Prefer `unknown` over `any` for values you haven't checked yet. `any` disables checking entirely; `unknown` forces a narrow before use.

Object shapes: type vs interface

Syntax	Behavior
<code>interface X { }</code>	Can merge, can extend
<code>type X = { }</code>	Can't merge, can union

Both describe an object's shape. `interface` declarations with the same name in the same scope merge into one; `type` aliases can't merge but can express unions and tuples that `interface` can't.

```
interface User {  
  id: string;  
}  
  
type Id = string | number;
```

Gotcha: two `interface User { ... }` declarations anywhere in your program silently merge into one. A stray declaration in a `.d.ts` file can widen your type with no visible edit to your code.

Unions, literals & narrowing

Guard	Narrows to
<code>typeof x === "string"</code>	<code>string</code>
<code>x instanceof C</code>	<code>C</code>
discriminant field	one union member

A union narrows to a specific member inside a guarded branch.

Discriminated unions — a shared literal field — narrow on a `switch` and scale better than a chain of `typeof` checks.

```
type Shape =  
  | { kind: "circle"; r: number }  
  | { kind: "square"; side: number };  
  
function area(s: Shape): number {  
  switch (s.kind) {  
    case "circle":  
      return Math.PI * s.r ** 2;  
    case "square":  
      return s.side ** 2;  
  }  
}
```

Gotcha: `typeof null === "object"`, so a `typeof x === "object"` guard alone doesn't exclude `null`. Skipping the check is exactly what triggers TS18048, “possibly undefined” — check for `null` explicitly first.

Functions

Modifier	Meaning
<code>x?: T</code>	Optional parameter
<code>...x: T[]</code>	Rest parameter
<code>x: T = v</code>	Default value

Overloads (multiple call signatures for one implementation) exist but are rare in application code — most functions need only one signature.

```
function greet(name: string, loud?: boolean) {  
    return loud ? `${name.toUpperCase()}!` : name;  
}
```

Generics

Syntax	Meaning
<code><T></code>	Type parameter
<code><T extends U></code>	Constrained parameter
<code><T = D></code>	Default type

A type parameter is to a type what a function parameter is to a value — the caller supplies it, often inferred from an argument.

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

Gotcha: constraining a type parameter before it's needed throws away inference for no benefit. Add `extends` only once a real usage requires it.

Utility types

Type	Result
<code>Partial<T></code>	All properties optional
<code>Pick<T, K></code>	Subset of keys <code>K</code>
<code>Omit<T, K></code>	<code>T</code> minus keys <code>K</code>
<code>Record<K, V></code>	Object with keys <code>K</code> , values <code>V</code>
<code>ReturnType<F></code>	<code>F</code> 's return type

Each one transforms an existing type rather than declaring a new one from scratch — reach for these before writing a shape by hand.

```
interface User {  
  id: string;  
  name: string;  
}  
  
type UserPreview = Pick<User, "id">;
```

Gotcha: `Omit<T, K>` doesn't validate that `K` is actually a key of `T`. A typo in the key silently does nothing instead of erroring.

Literal types, `as const` & `satisfies`

Syntax	Effect
<code>as const</code>	Literal type, deeply readonly
<code>satisfies T</code>	Check against <code>T</code> , keep the narrow type
<code>as T</code>	Unchecked assertion

`satisfies` validates a value against a type without widening it; `as` is an assertion that skips checking entirely.

```
const a = "GET" as const; // "GET"

const routes = {
  home: "/",
} satisfies Record<string, string>;
```

Warning: `as` compiles even when it shouldn't help you sleep at night — `{}` `as` `{ id: number }` type-checks, and `.id` reads as `number` right up until it's undefined at runtime.

Advanced type-level programming

Syntax	Meaning
<code>T extends U ? A : B</code>	Conditional type
<code>infer U</code>	Capture a type in a conditional
<code>{ [K in keyof T]: ... }</code>	Mapped type

This goes deeper than a cheat sheet should — conditional and mapped types compose into real complexity fast. The Handbook’s “Conditional Types” chapter is worth reading in full before writing your own.

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

Gotcha: a deeply recursive conditional type eventually hits TS2589 , “type instantiation is excessively deep.” Simplify the recursion rather than fighting the compiler.

Modules & strict config essentials

Setting	Effect
<code>import type { X }</code>	Type-only import, erased
<code>verbatimModuleSyntax</code>	Forces explicit import type
<code>noUncheckedIndexedAccess</code>	<code>arr[i]</code> includes undefined

Type-only imports never reach the compiled output.

`verbatimModuleSyntax` makes that explicit at the import site, which is what catches an accidental runtime import of a type-only module.

```
interface User {  
  id: string;  
}  
  
export type { User }; // erased – types only
```

Note: using a type-only import as a value (calling it, not just annotating with it) is TS1361 — the fix is almost always to import the runtime value separately.

Further reading

- [TypeScript Handbook](#)
- [TypeScript 5.9 release notes](#)