TypeScript

Version 5.9 · verified 2026-07-25

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

Contents

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
`let x: string`	Explicit annotation
`let x = "hi"`	Inferred as `string`
`unknown` vs `any`	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
`interface X { }`	Can merge, can extend
`type X = { }`	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.

```ts
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
`typeof x === "string"`	`string`
`x instanceof C`	`C`
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
`x?: T`	Optional parameter
`...x: T[]`	Rest parameter
`x: T = v`	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
`<T>`	Type parameter
`<T extends U>`	Constrained parameter
`<T = D>`	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
`Partial<T>`	All properties optional
`Pick<T, K>`	Subset of keys `K`
`Omit<T, K>`	`T` minus keys `K`
`Record<K, V>`	Object with keys `K`, values `V`
`ReturnType<F>`	`F`'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
`as const`	Literal type, deeply `readonly`
`satisfies T`	Check against `T`, keep the narrow type
`as T`	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
`T extends U ? A : B`	Conditional type
`infer U`	Capture a type in a conditional
`{ [K in keyof T]: … }`	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
`import type { X }`	Type-only import, erased
`verbatimModuleSyntax`	Forces explicit `import type`
`noUncheckedIndexedAccess`	`arr[i]` 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