TypeScript Type System

1. Foundations

three declaration spaces — value (const/let/var/function/params), type (interface/type/type params), namespace (dotted access A.B.C) — one name can occupy more than one space at once, e.g. a class is both value and type

structural typing — TS compares shapes, not declared names — but two same-shape classes with private members are NOT interchangeable

type erasure — types exist compile-time only; instanceof works against a class (real JS value) but errors (TS2693) against an interface

keyof T — union of T's key names; keyof(A|B) = shared keys only, keyof(A&B) = all keys

generic constraints <T extends U> — bounds what T must satisfy — a method spec { length(): number } differs from a property spec length: number

generic defaults <T = X> — a required type parameter may never follow a defaulted one

<K extends keyof T> — typed key parameter — the derived value T[K] goes in the value slot, not the header

indexed access T['k'] / T[number] — pulls a property's/element's type out; a union key T['a'|'b'] gives the union of those value types

typeof type query — value -> type, e.g. ReturnType<typeof fn> — distinct from runtime typeof narrowing below

generic class Box<T> — the call site is the spec; constructors and methods must honor the T it fixes

enum vs literal union — no = ever follows the enum name (only members use =); a literal union usually beats a numeric enum

typeof narrowing (values) — runtime typeof checks narrow a union at the value level — distinct from the type-space typeof query above; typeof returns one of: "string", "number", "boolean", "undefined", "object", "function", "symbol", "bigint"

string -> number conversion — canonical is Number(x); parseInt/parseFloat parse only a leading prefix, ignoring trailing garbage

checking if a string is numeric — !isNaN(Number(x)) — global isNaN(x) alone coerces first, so isNaN("") is false and isNaN("abc") is true

generics <T> on a function — keeps a signature value-agnostic instead of hardcoding one type

union types A | B — the value could be either — narrow before using members not common to both

assignability direction — narrower assigns to wider, never the reverse — string | undefined is not assignable to string

intersection A & B — combines members; a property with conflicting types across A and B collapses to never

interface vs type — interface X = {...} is illegal (interfaces never use =); interfaces merge by declaration, type aliases never do

literal types — a single exact value as a type; let widens to the general type (string), not assignable back to the literal

union of literals — 'a' | 'b' | 'c' — keep every member with |

rest vs spread — same ..., opposite directions — a new name next to ... is gathering (rest); an existing value next to ... is spreading

1a. Type erasure

which declarations emit JS — const / class / enum / namespace emit real runtime code; type, interface, and pure type syntax erase completely

instanceof needs a value — works against a class (real constructor); errors TS2693 against an interface (no runtime representation)

what a runtime check CAN interrogate — typeof / instanceof / in can only test real JS values and shapes, never a type

a generic type parameter at runtime — T itself is not a value — new T() and instanceof T both fail; pass a real constructor in instead

enum emits a real object — numeric enums map both directions (name<->value); string enums map name->value only

discriminated union tag is a real property — no type exists at runtime, so a switch can only branch on an actual property, never a type

namespace emits a value — a namespace compiles to a real runtime object

private/protected/readonly vs # — private/protected/readonly are compile-time only (still a readable JS property); # is the one real runtime-private mechanism

as and ! perform no check — both are purely compile-time — neither runs code nor verifies anything at runtime

declare and .d.ts describe, never create — referencing an absent declared value throws ReferenceError, not a silent undefined

emitDecoratorMetadata — the deliberate partial de-erasure that lets NestJS-style DI resolve classes (never interfaces) as tokens at runtime

1b. Nominal behavior

instanceof behaves nominally — walks the real prototype chain (actual constructor identity) — a structurally-identical plain object fails it even though it satisfies the class's shape

branded / nominal types — string & {__brand} or unique symbol — not assignable from the plain type; a brand does not survive serialization

private/protected members — a class with a private/protected member is only assignable from that exact declaration — the one exception structural typing carves out for classes

string enums — TS's one genuinely nominal built-in type — a plain string literal like 'active' is not assignable to the enum type

2. any / unknown / never

unknown (safe top type) — accepts anything in, blocks all use until narrowed — even typeof === 'object' has a null hole

any (escape hatch, also a top type) — unchecked in both directions; unknown is the safer default — narrow before use

never (bottom type) — the type of a value that can't exist; T | never = T, T & never = never

exhaustiveness check (const _: never = x) — assigning the leftover branch to a never-typed variable fails to compile if a union case was missed

{} — the non-nullish top type — accepts everything except null/undefined; object rejects primitives

3. Narrowing & discriminated unions

tagged union + narrow on the tag — switch/if on the shared literal tag; only members common to every branch are readable before narrowing

narrowing via switch + exhaustiveness — falling off the end yields undefined; the error only appears if the declared return type rejects it

instanceof / in as guards — 'key' in x narrows by property presence — no tag needed

checking if a value is an array — Array.isArray(x) — typeof x on an array is just "object", it can't tell arrays apart

user-defined guard x is T — a function returning x is T; narrows the caller's variable after a truthy check — the key in x is T must be quoted

type guard inside .filter — TS 5.5+ infers the predicate automatically — no x is T annotation needed

narrow unknown in catch — catch bindings are unknown under strict mode; a throw can deliver a non-Error, so guard before use

4. Assertions

as type assertion — legal only either-direction between the two types; compiles but lies at runtime — a as Dog still has no real .breed

as const — deep readonly at every depth; value-space only — illegal on the right of type X = ...

non-null assertion x! — emits nothing at runtime; a wrong ! just fails later at the property read

satisfies operator — checks the value against a type but keeps the narrower inferred type — no widening

double assertion as unknown as T — bypasses the overlap check entirely; nothing is checked or converted at runtime

5. Utility types

Partial<T> / Required<T> — make every property optional / make every property required

Pick<T,K> / Omit<T,K> — select or drop named keys — usable as a return type, not just inline

Record<K,V> — a literal-union key makes every key required; a plain string key stays open (partial)

Readonly<T> / readonly T[] / tuples — shallow only — one level deep, not recursive

ReturnType<T> — extracts a function type's return type; needs typeof on a value first to get its type

Parameters<T> — extracts a function type's parameter list as a tuple

Awaited<T> — recursively unwraps nested Promises

Exclude / Extract / NonNullable — Exclude removes matching union members, Extract keeps only matching ones, NonNullable drops null/undefined

6. Mapped types

{ [K in keyof T]: ... } — rebuilds an object type by iterating its keys; the homomorphic form passes modifiers through unchanged

modifiers ? / readonly / -? / -readonly — add or strip optional/readonly per key

key remapping as in mapped types — { [K in keyof T as NewKey]: ... } renames keys, or filters one out entirely when the as clause resolves to never

7. Conditional types

T extends U ? X : Y — a type-level if/else based on assignability

distributive over unions — a naked type parameter in the check position distributes over each union member; over never it collapses to never

distribution needs a NAKED type parameter — wrapping it in a tuple, [T] extends [U], turns distribution off

recursive conditional & mapped types — a conditional/mapped type can recurse to transform nested structures, e.g. DeepReadonly<T>

infer — placement & multiple sites — introduces a type variable inside the extends clause to capture part of the matched type

infer (extract a type) — the standard way to pull one piece — an element, a return type — out of a larger type

8. Template-literal types

`${A}-${B}` template-literal types — build string literal types like template strings; distributes over unions in each slot (cross product)

Uppercase / Lowercase / Capitalize / Uncapitalize — the built-in intrinsic type-level string transforms

9. Function types

function overloads — resolution — first matching signature (declaration order) wins; return types never merge or union

call signature — the braces form — type F = { (x: string): number } — a callable value type; add named members for a callable object

construct signature — new (...args) => T types a constructor itself; abstract new (...) => T types an abstract class

the match-all function type — (...args: never[]) => unknown accepts any function; the unknown[] version rejects typed-parameter functions by contravariance

type parameter position with a callback — a shared <T> can sit outside, in the callback's return, or in the callback's own parameter — the parameter slot is easiest to miss

generic wrapper around a function — where the hole goes — put <T> on the piece that should vary, not the whole shape — <T extends (...args: any[]) => any> only proves 'is a function'

inference through a contravariant callback parameter — candidates from a callback parameter intersect, not union — string and number there infers T = string & number

overloads collapsing into one generic — a verified exception — a rest-parameter generic over a union of tuple shapes, <T extends [Date,Date]|[number]> (...args: T), can legally replace overloads that looked irreducible

composing functions — typing pipe/compose — a generic pipe/compose signature threads one type variable through every stage

10. Classes

readonly / param properties — readonly blocks reassignment at compile time only — it never freezes the value at runtime

public / private / # — private/protected are compile-time only and still ordinary readable JS properties; # is genuinely private at runtime

implements — a pure compile-time contract check — zero runtime effect, and a mismatch errors at the class name

method vs arrow-function field — m() {} binds this at the call site (detachable, can lose it); m = () => {} binds this once per instance (safe as a passed-around callback, costs one field per instance) — prefer the arrow field for event handlers

get/set accessors — get x(): T / set x(v: U) — a getter with no setter is externally readonly; TS 4.3+ allows the setter's parameter type to differ from the getter's return type; the pair counts as one member for an interface

12. Runtime validation

excess-property check (literal vs parsed) — only fires on a fresh object literal assigned directly — a variable or already-parsed value skips it

parse, don't validate — return the narrowed value itself from a parser, not just a boolean — a guard alone strips/transforms nothing

derive the type from the schema — type X = z.infer<typeof schema> — infer needs typeof on the schema value, and it's a generic <>, never a call ()

throw vs result (parse / safeParse) — .parse throws on failure; .safeParse returns a discriminated Result instead

unknown-key policy — a schema library must decide: strip, reject, or keep keys it didn't declare

TS has no exact object type — "no extra keys" isn't expressible structurally — extras are always assignable except the excess-property check on a fresh literal

Checked<T> — a wrapper type only producible by actually running validation, so an unvalidated value can't type-check as trusted

never spread into persistence — map fields explicitly rather than spreading an object into a DB write, so extra/renamed fields can't sneak through

boundaries — where untyped data enters — HTTP request bodies; third-party API responses; queue/workflow inputs; event/pub-sub messages; file uploads; config/env vars; database reads

where validation lives — parse once at the edge; everything downstream trusts the typed value

coercion: strings in, typed values out — schema coercion turns raw string input into typed output — input and output types can legitimately differ

schema composition mirrors utility types — zod's .omit takes a mask object, not a string, the same shape as the Omit<T,K> utility type

13. Variance & assignability

arrays are covariant (and unsound) — Dog[] assigns to Animal[], which lets an unsound push through the wider reference — the fix is readonly T[]

function parameters are contravariant — a function accepting a WIDER parameter type is assignable where a narrower one is expected

method params are bivariant (the deliberate hole) — method-shorthand syntax checks parameters loosely both ways (unsound, kept for practicality); property-syntax stays strictly contravariant

return types are covariant — a function returning a NARROWER type is assignable where a wider return is expected

ReadonlyArray<T> vs T[] — a one-way door — T[] assigns into ReadonlyArray<T> freely; the reverse is always rejected

tuples and variance — a tuple assigns into a compatible array type; the reverse (array into tuple) is rejected

Map<K,V> covariant in V — because its lib method signatures use shorthand (bivariant) syntax, not strict variance

in / out variance annotations — declare a generic's variance explicitly and let the compiler check it — violation is TS2636

keyof T is contravariant in T — a wider T produces a narrower keyof T (fewer guaranteed keys) — the direction inverts

mutable object property covariance — the same unsound hole as arrays — a narrower-typed object aliased through a wider reference lets you write back an incompatible value

your own generic — measure where T sits — where T appears (return-only / parameter-only / both / unused) decides covariant, contravariant, invariant, or bivariant

the phantom type — an unused T is bivariant — a type parameter that never appears in any member behaves bivariantly, since nothing structurally constrains it

void return accepts any return type — a callback typed to return void still accepts a function that returns something — the value is just ignored

parameter count — a function needing FEWER parameters than expected is assignable; needing MORE is not

tuple arity vs element variance — length compatibility and per-element variance are checked as separate, independent rules

optional is presence, not variance — an optional property changes whether the key must exist, not the compatibility rules for its value type

readonly on a property is not checked — invisible to assignability for plain properties — only readonly arrays get a dedicated check, TS4104

never and unknown as the two edges — any is assignable to everything except never; unknown accepts everything and is assignable to almost nothing

T[K], index signatures, construct signatures — variance applies at each position independently — an index-signature read can be unsound without noUncheckedIndexedAccess

higher-kinded types and the workarounds — a bare type constructor F<_> as a parameter isn't legal TS (TS2315); libraries like fp-ts encode it indirectly via a Kind pattern

14. Inference

widening: let vs const on a literal — let widens a literal to its general type; const keeps the literal type

why an object literal's property widened — a plain property inside an object literal widens the same way let would, unless fixed by as const, an explicit type, or context

when a union widens (and when it collapses) — let widens every member of an inferred union; TS also collapses redundant subtypes into a wider member

contextual typing — inference flows from the expected type at a position (e.g. a callback parameter's type comes from where it's passed)

inference from a generic call site — TS infers one T per call by picking a best-fit candidate — it does not union multiple candidates together

generic vs overload — the decision — collapse overloads into one generic only when the return type doesn't change shape between forms

groupBy that preserves key literals — the key type should be inferred from the callback's return (K extends string), not from keyof T

NoInfer<T> — marks a position that participates in the check but is excluded from what fills the type parameter

declaration merging — same-named interfaces merge their members; type aliases never merge

constraint vs inference — two different jobs — a constraint (extends) only gates which arguments are legal — it does not narrow what gets inferred for T

when a constraint DOES change what's inferred — a primitive constraint like T extends string turns off literal widening for that argument, but not for arrays

keep literals & tuples at a call site — three ways: as const at the call site, a <const T> type parameter, or an explicit narrow annotation

life of a type parameter — fill order — explicit <> first, then arguments, then the expected result type, then a default, then unknown last

15. Library-grade authoring

derive union from array (typeof ARR[number]) — turns a const array's element type into a union without repeating the literals

build a small utility type from scratch — hand-roll a Pick/Omit-shaped type to prove the mapped/conditional mechanics, not just recognize the built-in

variadic tuples [...T] and [H, ...infer R] — spread/destructure tuple types like array patterns — infer stays lowercase, a dropped slot still needs a real type, not a made-up name

assertion functions: asserts x is T — narrows its argument by throwing instead of returning a boolean — only narrows when called through an explicitly-typed reference (TS2775)

const type parameters <const T> — infers the literal/tuple type as given instead of widening it

abstract new (...) => T — types an abstract class itself, not just its instances — typeof SomeClass is a class name used as a value/parameter type

16. Vocabulary

assignability — the core question every rule answers: can a value of type A be used where type B is expected?

widening — a literal/narrow type relaxing to its general type once nothing anchors it narrow

narrowing — shrinking a union to a smaller set of members via a runtime check

variance — the umbrella term for how subtyping of T affects subtyping of a type built from T, e.g. Box<T>

covariant — subtyping direction is preserved: Dog <: Animal implies Box<Dog> <: Box<Animal>

contravariant — subtyping direction flips: a function accepting Animal is a subtype of one accepting Dog

invariant — neither direction holds — Box<Dog> and Box<Animal> are unrelated unless T matches exactly

bivariant — accepted both ways unsoundly — TS's deliberate hole for method-shorthand parameters

distributive — a conditional type that, given a naked union type parameter, applies itself to each member separately

homomorphic — a mapped type that iterates keyof T and structurally mirrors T's own modifiers (readonly/optional) instead of a hardcoded literal-key spec

discriminant — the shared literal-typed property (a 'tag') that lets a switch/if narrow a union without instanceof

contextual typing — a type inferred from the position/slot a value is used in, rather than from the value alone

declaration merging — same-named interface declarations combine into one; type aliases and classes never merge this way

higher-kinded type — a type constructor taking a type constructor as its own parameter (e.g. F<_>) — not directly expressible in TS

phantom type — a type parameter that never appears in any member — carries no runtime data, exists purely to tag the type

opaque type / branding — a type made distinguishable from its underlying primitive by an unused marker field, so plain values can't slip in unchecked