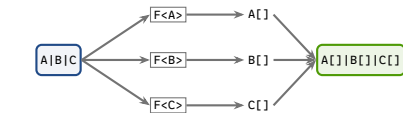


In one line: Type parameters are **solved by inference** at each use, bounded by extends; **conditional, mapped** and **template-literal** types form a small pure functional language the checker evaluates — erased at runtime, limited by instantiation depth, and only as good as the inference it enables at call sites.

1 distributive conditional

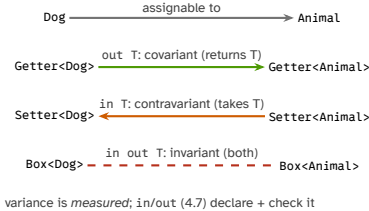
$F<T> = T \text{ extends any? } T[] : \text{never}$



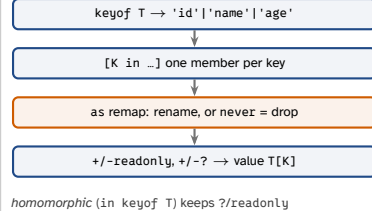
$[T] \text{ extends } [any]: \text{no fan-out} \rightarrow (A|B|C)[]$

only a naked T distributes; $\text{never} = \text{empty union} \rightarrow \text{never}$
Exclude, Extract, NonNullable-style filters rely on it

2 variance: does <> keep the arrow?



3 a mapped type, step by step



How it works

- Constraint** $\langle T \text{ extends HasId} \rangle =$ upper bound, like Swift T : Protocol but *structural*; default $\langle T = \text{string} \rangle$. Rule: a type parameter must appear **twice** (links input to output), else use unknown.
- Inference** collects candidates from each argument; literals widen unless $T \text{ extends string}$ (keeps 'a') or $\langle \text{const } T \rangle$ (5.0 — infers as if as const ; arrays need a $\text{readonly unknown}[]$ constraint). $\text{NoInfer}\langle T \rangle$ (5.4) removes one position from inference.
- keyof** T : union of keys (index signature string gives $\text{string}|\text{number}$). $\text{typeof } v: \text{value} \rightarrow \text{type}$. $T[K]$: indexed access; $T[\text{number}] =$ element type.
- Conditional** $T \text{ extends } U ? X : Y$; $\text{infer } R$ captures a piece of the match. With T still generic it is **deferred** — inside a generic body you can't narrow it, hence casts in implementations.
- Template literals** (4.1): ``on${Capitalize<K>}``, cross-product over unions, infer parses strings. Intrinsic: Uppercase, Lowercase, Capitalize, Uncapitalize.
- Recursion**: $\text{type } \text{Json} = \dots | \text{Json}[] | \{[k: \text{string}]: \text{Json}\}$. Tail-recursive conditional types go ≈ 1000 deep (4.5); else error 2589 "instantiation is excessively deep and possibly infinite".
- Swift**: generics there are specialised/witness-table dispatched, invariant (except built-ins), and have no conditional/mapped types. some $P \approx$ a generic T ; any $P \approx$ plain interface type (no box needed: structural).

Utility types from scratch (lib.es5.d.ts)

```
export {}; // module scope: shadows the global names
type Partial<T> = { [K in keyof T]?: T[K] };
type Required<T> = { [K in keyof T]-?: T[K] };
type Readonly<T> = { readonly [K in keyof T]: T[K] };
type Pick<T, K extends keyof T> = { [P in K]: T[P] };
type Record<K extends keyof any, V> = { [P in K]: V };
type Exclude<T, U> = T extends U ? never : T; // distributes
type Extract<T, U> = T extends U ? T : never;
type Omit<T, K extends keyof any> = Pick<T, Exclude<keyof T, K>>;
type NonNullable<T> = T & {}; // since 4.8
type ReturnType<T extends (...a: any) => any> =
  T extends (...a: any) => infer R ? R : any;
type Parameters<T extends (...a: any) => any> =
  T extends (...a: infer P) => any ? P : never;
type Awaited<T> = // simplified: real one handles any thenable
  T extends PromiseLike<infer V> ? Awaited<V> : T;
```

Example — what seniors write

```
type Events = { login: { id: string }; logout: undefined };
type Handlers = { // onLogin, onLogout
  [K in keyof Events as `on${Capitalize<string & K>}`]:
    (e: Events[K]) => void };
function get<T, K extends keyof T>(o: T, k: K): T[K] {
  return o[k]; }
type Params<S> = S extends `${string}:${infer P}/${infer R}`
  ? P | Params<`${R}`>
  : S extends `${string}:${infer P}` ? P : never;
type P1 = Params<`/u:/id/post/:pid`>; // 'id' | 'pid'
type DeepReadonly<T> = T extends (...a: any[]) => any ? T
  : { readonly [K in keyof T]: DeepReadonly<T[K]> };
function fsm<S extends string>(states: S[], start: NoInfer<S>) {}
fsm(['on', 'off', 'of']); // error thanks to NoInfer
interface Getter<out T> { get: () => T }
interface Setter<in T> { set: (v: T) => void }
```

Interview traps

- ! `{a:1}` is assignable to the constraint of T , but T could be instantiated with a different subtype — you returned a *new* object where the caller's exact T was promised.
- ! `Omit/keyof` on a *union* keep only common keys: distribute first — $T \text{ extends any? } \text{Omit}\langle T, K \rangle : \text{never}$.
- ! $\text{IsNever}\langle T \rangle = T \text{ extends never? } \dots$ yields *never* for *never*: write $[T] \text{ extends } [\text{never}]$.
- ! Arrays are covariant only through method bivariance: $\text{Dog}[]$ as $\text{Animal}[]$ then $\text{push}(\text{cat})$ compiles.
- ! Clever types cost compile time and error readability; prefer overloads or a plain union when a call site doesn't gain inference.

Remember

Constrain, infer, then compute: conditional = if, mapped = for-each, infer = pattern bind, recursion = loop.

Likely questions

1. Distributive conditional? — a naked T maps over each union member; wrap in $[T]$ to stop it.
2. Covariant vs contravariant? — outputs keep the subtype arrow, inputs flip it, both = invariant.
3. Why NoInfer ? — so a default/secondary argument can't widen T .
4. const type parameter? — literal/readonly inference without as const at every call.
5. Implement $\text{Pick}/\text{ReturnType}$? — see the block above.