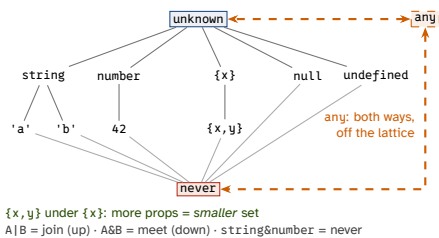
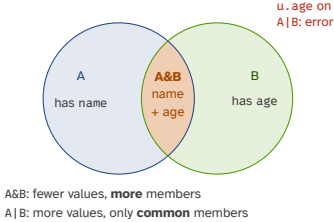


In one line: A type is a **set of values**; assignability is **subset**, judged by **shape** (structural), not by name. Unions must be **narrowed** along control flow before use — and it is all **erased** at runtime. Soundness is traded for ergonomics in known holes: any, as, method bivariance, readonly props.

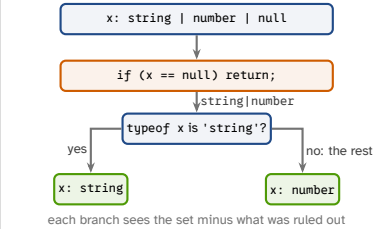
1 the lattice (up = bigger set)



2 values, not members



3 control-flow narrowing



How it works

- **Structural:** `{x, y}` fits wherever `{x}` is expected (Swift: **nominal**). `private/#private` fields make a class effectively nominal.
- **Excess-property check** — the exception: a *fresh* object literal assigned straight to a typed target may not carry unknown keys (typo catcher); via a variable it passes.
- **interface vs type:** interfaces **merge** (re-declare = augment globals, lib types); extends reports conflicts *at the declaration* and is cached by name (cheaper than big &). Only type does unions, tuples, mapped/conditional. & turns a conflicting prop into `never`.
- **Narrowing:** `typeof` ("object" includes `null`!), `instanceof` (prototype chain), `"k" in x`, equality (`==` `null` catches `null` and `undefined`; tag compare), truthiness (also drops `0`, `"", NaN`). Assignments re-widen; a reassigned `let` loses narrowing in callbacks (5.4: kept after the last assignment).
- **Guards:** `(x): x is Fish` — the body is *trusted, not checked*. asserts `x is T` / asserts `cond` narrows *after* the call. 5.5 infers predicates: `arr.filter(x => x !== undefined)`.
- **Top/bottom:** `unknown` takes anything, gives nothing until narrowed. `never` = empty set: assignable to all, nothing to it. any switches checking off *and spreads*. `void` = "result ignored": `a()` => `void` slot accepts `()` => `number`.
- **Widening:** `const s = 'a'` is `'a'`; `let` and object props under to `string`, as `const` = deep `readonly` + literals.
- **Optionality:** `a?: T` may be *absent*; `a: T|undefined` must be *present*. `exactOptionalPropertyTypes` forbids writing `a: undefined` into `a?: T`.
- **Functions:** fewer params is fine (`forEach(x => ...)`); `return` is covariant; params contravariant under `strictFunctionTypes` — *except method syntax* `m(x: T): void`, kept **bivariant** (so `Dog[]` fits `Animal[]`).

Example — discriminated union, guards

```
type Shape = // Swift: enum + associated values
  | { kind: 'circle'; r: number }
  | { kind: 'rect'; w: number; h: number };
function area(s: Shape): number {
  switch (s.kind) { // the tag narrows the union
    case 'circle': return Math.PI * s.r ** 2;
    case 'rect':   return s.w * s.h;
    default: return s satisfies never; // new kind -> error
  }
}
const isStr = (x: unknown): x is string => typeof x === 'string';
function check(c: unknown, msg: string): asserts c {
  if (!c) throw new Error(msg);
}
```

annotation · satisfies · as

```
const r1: Record<string, string> = { home: '/' };
r1.hmoe; // compiles: the annotation IS the type now
const r2 = { home: '/' } satisfies Record<string, string>;
r2.hmoe; // error: inferred keys kept (TS 4.9)
const r3 = {} as Record<'home', string>; // compiles, lies
interface P { x: number }
const p: P = { x: 1, y: 2 }; // error: excess property
const t = { x: 1, y: 2 }; const q: P = t; // OK: structural
```

Swift → TS

Swift	TypeScript
protocol conformance (nominal)	any matching shape (structural)
enum + associated values	union with a literal kind tag
compiler-exhaustive switch	<code>const x: never</code> in default
<code>Optional<Optional<T>></code> nests	<code>T undefined undefined</code> collapses
<code>as!</code> traps at runtime	<code>as</code> never checks anything
<code>Any</code>	<code>unknown</code> (not <code>any</code>)

Interview traps

- ! `{}` means "any non-nullish value" (even `5`), not "empty object"; `object` = non-primitive; `empty` = `Record<string, never>`.
- ! `{readonly x}` is assignable to `{x}` (hole); but `readonly T[]` is not assignable to `T[]`. `readonly` is shallow and compile-time only.
- ! A wrong `x is T` guard is an unchecked lie — worse than `as`, it hides at every call site.
- ! Callbacks declared as methods are bivariant: write `onX: (e: E) => void` (property) to get strict checks.
- ! `if (count)` also drops `0`: compare with `undefined`.
- ! `as` refuses only types that don't overlap at all; `x as unknown as T` forces anything — grep for it in review.

Remember

Types are sets · assignable = subset · shape, not name · narrow before you touch.

Likely questions

1. type or interface? — interface for object contracts; type for unions and computed types.
2. `unknown` vs `any`? — both accept all; `unknown` forces a narrow before use, any silently disables checking downstream.
3. `satisfies` vs `as`? — validates yet keeps the inferred type; `as` asserts, checked only for "sufficient overlap".
4. Exhaustiveness? — assign the leftover to `never`.
5. Why is an extra key rejected only sometimes? — the excess-property check fires on *fresh* literals only.