

TS

Frontend · most asked

TypeScript interview questions

25 questions · 6 easy · 15 medium · 4 hard. Open one to see the answer, a code example and the likely follow-up, then mark it as known or to review.

1

What is TypeScript and why use it over JavaScript?

Easy



#basics

TypeScript is JavaScript plus a **static type system**. It compiles to plain JavaScript, and **types are erased at runtime**: they only exist while you write and build code.

Benefits:

- many bugs (typos, wrong arguments, **undefined** access) are caught before the code runs
- much better editor support: autocomplete, inline docs, safe renames
- types act as documentation and make refactoring large codebases safer

Costs: a build step, some learning curve, and the fact that types cannot validate data coming from outside (API responses, user input) at runtime.

Likely follow-up: Since types are erased, how do you validate an API response at runtime?

2

What is the difference between **interface** and **type**?

Easy



#types #interfaces

For describing object shapes they are almost interchangeable. The differences:

- **type** can name **any** type: unions (**'a' | 'b'**), tuples, primitives, mapped and conditional types. **interface** only describes object shapes.
- **interface** supports **declaration merging**: declaring it twice merges the members (used to extend library types like **Window**). A **type** cannot be re-opened.
- **interface** extends with **extends**; **type** composes with intersections (**&**). **extends** reports conflicts more clearly.

A common rule: `interface` for object shapes and public APIs, `type` for unions and everything else. Consistency within a codebase matters more than the choice.

Example

```
interface User { id: number; name: string }
interface Admin extends User { permissions: string[] }

type Status = 'active' | 'banned' // only a type can do this
type Member = User & { status: Status }

const a: Admin = { id: 1, name: 'Asha', permissions: ['delete'] }
const m: Member = { id: 2, name: 'Ben', status: 'active' }
console.log(a.permissions, m.status)
```

3

What is the difference between `any`, `unknown` and `never`?

Medium 🔥🔥 #types #safety

- `any` switches type checking off. Anything can be assigned to it and you can do anything with it, so errors spread silently. Avoid it.
- `unknown` is the **safe** version: anything can be assigned to it, but you cannot use it until you narrow it (`typeof`, `instanceof`, a type guard). Use it for `JSON.parse` results, `catch` errors and other untrusted input.
- `never` is the type with **no values**: a function that always throws, or a branch that can never happen. It is used for exhaustiveness checks.

`void` is different again: a function returns nothing useful.

Example

```
function parse(json: string): unknown {
  return JSON.parse(json)
}

const data = parse('{ "name": "Asha" }')
// data.name // error: 'data' is of type 'unknown'
if (typeof data === 'object' && data !== null && 'name' in data) {
  console.log((data as { name: string }).name)
}

function fail(msg: string): never {
  throw new Error(msg)
}

try { fail('boom') } catch (e) { console.log(e instanceof Error ?
e.message : e) }
```

4

What are generics? Give an example.

Medium 🔥🔥 #generics

Generics are **type parameters**: they let a function, class or type work with many types while keeping the relationship between input and output.

Without generics you would write `any` and lose the type information. With `function first<T>(arr: T[]): T | undefined`, calling it with `string[]` returns `string | undefined`.

TypeScript usually **infers** `T` from the arguments. You can restrict it with a **constraint** (`T extends { id: number }`) and give it a default (`T = string`).

Everyday examples: `Array<T>`, `Promise<T>`, `useState<T>`, and typed API helpers like `fetchJson<T>(url)`.

Example

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

function findById<T extends { id: number }>(items: T[], id: number): T |
undefined {
  return items.find((item) => item.id === id)
}

const n = first([1, 2, 3])           // number | undefined
const s = first(['a', 'b'])          // string | undefined
const user = findById([{ id: 1, name: 'Asha' }], 1)
console.log(n, s, user?.name)       // name is known: T kept the full
shape
```

? Likely follow-up: What does `<T extends keyof U>` mean?

5

Which built-in utility types do you use most?

Medium 🔥🔥 #utility types

- `Partial<T>`: all properties optional (update payloads)
- `Required<T>`: all properties required
- `Readonly<T>`: all properties read-only
- `Pick<T, 'a' | 'b'>`: keep only some keys
- `Omit<T, 'password'>`: drop some keys
- `Record<K, V>`: an object with keys `K` and values `V`
- `ReturnType<typeof fn>` and `Parameters<typeof fn>`: types taken from a function
- `Awaited<T>`: the resolved type of a promise
- `NonNullable<T>`, `Exclude<U, X>`, `Extract<U, X>`: filter unions

They let you **derive types from one source of truth** instead of copying them, so a change to `User` flows everywhere.

Example

```
interface User { id: number; name: string; email: string; password: string }

type PublicUser = Omit<User, 'password'>
type UserUpdate = Partial<Pick<User, 'name' | 'email'>>
type Roles = Record<'admin' | 'viewer', string[]>

const pub: PublicUser = { id: 1, name: 'Asha', email: 'a@x.io' }
const patch: UserUpdate = { name: 'Asha K' }
const roles: Roles = { admin: ['*'], viewer: ['read'] }
console.log(pub, patch, roles.admin)
```

? Likely follow-up: How would you implement `Partial` or `Pick` yourself?

6

What is type narrowing? What is a type guard?

Medium



#narrowing #type guards

Narrowing is how TypeScript refines a broad type (like `string | number`) to a more specific one inside a branch, by following your runtime checks:

- `typeof x === 'string'`
- `x instanceof Date`
- `'swim' in animal`
- equality and truthiness checks (`if (user)`)
- discriminant checks (`shape.kind === 'circle'`)

A **user-defined type guard** is a function whose return type is a **type predicate** like `value is Fish`. When it returns `true`, TypeScript narrows the argument. Assertion functions (`asserts value is string`) narrow by throwing instead.

Be careful: the compiler trusts your guard, so a wrong implementation lies to it.

Example

```
type Fish = { swim: () => string }
type Bird = { fly: () => string }

function isFish(pet: Fish | Bird): pet is Fish {
  return 'swim' in pet
}

function move(pet: Fish | Bird) {
  return isFish(pet) ? pet.swim() : pet.fly() // narrowed in each
  branch
}

function format(x: string | number) {
  return typeof x === 'number' ? x.toFixed(2) : x.toUpperCase()
}

console.log(move({ swim: () => 'swimming' }), format(3.14159),
format('hi'))
```

7

What is a discriminated union, and how do you make a `switch` exhaustive?

Medium



#unions #narrowing

A discriminated union is a union of object types that all share a **literal "tag" property**, such as `kind` or `status`. Checking the tag narrows the whole object, so each branch knows exactly which fields exist.

It is the standard way to model state: `{ status: 'loading' } | { status: 'success'; data: T } | { status: 'error'; error: string }`. Impossible states (data and error at once) cannot be expressed.

For **exhaustiveness**, assign the value to `never` in the `default` branch. If someone adds a new variant and forgets to handle it, **the code stops compiling**.

Example

```
type Shape =
  | { kind: 'circle'; radius: number }
  | { kind: 'rect'; width: number; height: number }

function area(s: Shape): number {
  switch (s.kind) {
    case 'circle': return Math.PI * s.radius ** 2
    case 'rect': return s.width * s.height
    default: {
      const unreachable: never = s // errors if a kind is not handled
      return unreachable
    }
  }
}

console.log(area({ kind: 'rect', width: 2, height: 3 }))
console.log(area({ kind: 'circle', radius: 1 }).toFixed(2))
```

8

What does `strict` mode do? Why does `strictNullChecks` matter so much?

Easy



#config #safety

`"strict": true` in `tsconfig.json` turns on a family of checks, including `noImplicitAny`, `strictNullChecks`, `strictFunctionTypes` and `strictPropertyInitialization`. New projects should always enable it.

`strictNullChecks` is the most important one. Without it, `null` and `undefined` are allowed in **every** type, so `user.name` compiles even when `user` might be missing. With it, `null` and `undefined` are separate types, and **you must handle them before use** (with a check, `?.`, or `??`).

That catches the single most common JavaScript runtime error: "cannot read properties of undefined".

Example

```
type User = { name: string }
const users: User[] = [{ name: 'Asha' }]

const found = users.find((u) => u.name === 'Ben') // User | undefined
// console.log(found.name)                       // error: possibly
// 'undefined'
console.log(found?.name ?? 'not found')

if (found) console.log(found.name)                // narrowed to User
```

🔗 Likely follow-up: What other `tsconfig` options would you turn on?
(`noUncheckedIndexedAccess`)

9

Enums vs union of string literals: which do you prefer?

Medium 🔥🔥 #enums #unions

A **string literal union** (`type Role = 'admin' | 'user'`) exists only at compile time: no runtime code, works naturally with JSON and APIs, and gives autocomplete.

An **enum** (`enum Role { Admin = 'admin' }`) generates a real JavaScript object, so you can iterate over its values. Downsides: it is one of the few TypeScript features that **emits runtime code**, numeric enums accept any number, and enums are not supported by type-stripping tools such as Node's built-in TypeScript support.

Many teams prefer unions, or an `as const` object when they also need the values at runtime. `const enum` is inlined but causes issues with isolated module builds.

Example

```
const ROLES = ['admin', 'editor', 'viewer'] as const
type Role = (typeof ROLES)[number] // 'admin' | 'editor' | 'viewer'

function can(role: Role) { return role !== 'viewer' }

console.log(ROLES.map((r) => `${r}: ${can(r)}`))

enum Direction { Up = 'UP', Down = 'DOWN' }
console.log(Direction.Up, Object.values(Direction))
```

10

What does it mean that TypeScript is structurally typed?

Medium



#types #basics

TypeScript compares types by their **shape**, not by their name or where they were declared. If an object has all the properties a type requires, it is assignable to that type, even if it never mentions it. This is sometimes called **duck typing at compile time**.

Consequences:

- two interfaces with the same members are interchangeable
- an object with **extra** properties is fine when passed through a variable, but an object literal written inline gets an "excess property" error, to catch typos
- classes are compared structurally too, unless they have **private** members

To make types like `UserId` and `OrderId` (both `string`) incompatible, you use **branded types**.

Example

```
interface Point { x: number; y: number }
function show(p: Point) { return `(${p.x}, ${p.y})` }

const p3 = { x: 1, y: 2, z: 3 }
console.log(show(p3))           // OK: has x and y
// show({ x: 1, y: 2, z: 3 })  // error: excess property 'z'

type UserId = string & { __brand: 'UserId' }
const id = 'u_42' as UserId
console.log(id)
```

11

Explain `keyof`, `typeof` (in a type position) and indexed access types.

Medium



#type operators

- `keyof T`: the union of the property names of `T`. For `{ id: number; name: string }` it is `'id' | 'name'`.
- `typeof value`: in a type position, it **turns a runtime value into its type**. Useful for config objects and functions (`ReturnType<typeof fn>`).
- `T[K]`: indexed access, the type of property `K` on `T`, for example `User['address']`. `T[number]` gives the element type of an array or tuple.

Combined, they let you write a type-safe `getProperty` whose return type depends on the key you pass.

Example

```
const config = { port: 3000, host: 'localhost', debug: false }
type Config = typeof config // { port: number; host:
string; debug: boolean }
type ConfigKey = keyof Config // 'port' | 'host' | 'debug'

function get<T, K extends keyof T>(obj: T, key: K): T[K] {
  return obj[key]
}

const port = get(config, 'port') // number
const host = get(config, 'host') // string
console.log(port + 1, host.toUpperCase())
// get(config, 'nope') // error: not a key of Config
```

12

What is the difference between an optional property and a property that can be `undefined`? What does `readonly` do?

Easy 🔥🔥 #types #immutability

`name?: string` means the key **may be missing**. `name: string | undefined` means the key **must be present**, but its value can be `undefined`. With

`exactOptionalPropertyTypes` on, the difference is enforced strictly.

`readonly` prevents re-assigning a property after creation. It is **compile-time only and shallow**: nested objects are still mutable, and nothing stops mutation at runtime.

For arrays, `readonly string[]` (or `ReadonlyArray<string>`) removes mutating methods like `push`. `Readonly<T>` makes every property of a type read-only, and `as const` makes a literal deeply read-only.

13

What are type assertions (`as`) and the non-null assertion (`!`)? When are they dangerous?

Easy 🔥🔥 #assertions #safety

A **type assertion** (`value as User`) tells the compiler "trust me, this is a `User`". It does no conversion and **no runtime check**. TypeScript only allows assertions between types

that overlap; `as unknown as X` forces anything.

The **non-null assertion** `user!.name` removes `null` and `undefined` from the type.

Both are dangerous because they silence the compiler: if you are wrong, you get exactly the runtime error TypeScript was meant to prevent. Prefer narrowing, type guards, or runtime validation (Zod) for external data. Reasonable uses: DOM queries you know succeed, and test code.

Not to be confused with `as const` or `satisfies`, which are safe.

14

What does `as const` do?

Medium



#literal types #inference

`as const` is a **const assertion**. It tells TypeScript to infer the **narrowest possible type**:

- string and number values become **literal types** (`'GET'` instead of `string`)
- object properties become `readonly`
- arrays become `readonly` tuples

It is commonly used for config objects, route tables and lists of allowed values, from which you derive a union type with `typeof X[number]` or `keyof typeof X`. That gives you the values at runtime and the type at compile time from one definition, which is often a cleaner alternative to an enum.

Example

```
const methods = ['GET', 'POST'] as const           // readonly ['GET', 'POST']
type Method = (typeof methods)[number]             // 'GET' | 'POST'

const routes = { home: '/', about: '/about' } as const
type RouteName = keyof typeof routes                // 'home' | 'about'

function go(name: RouteName, method: Method = 'GET') {
  return `${method} ${routes[name]}`
}
console.log(go('about'), methods.length)
```

15

What does the `satisfies` operator do, and how is it different from a type annotation?

Medium



#inference #modern ts

`satisfies` (TypeScript 4.9) **checks** that a value matches a type, but **keeps the more specific inferred type** of the value.

With an annotation (`const config: Record<string, string | number> = ...`), the variable becomes that wide type, so `config.port` is `string | number` and you lose the detail.

With `const config = { ... } satisfies Record<string, string | number>`, you still get an error for a wrong value or a typo, but `config.port` stays `number`.

Use it for config objects, theme tokens and lookup tables where you want both validation and precise types. It combines well with `as const`.

Example

```
type Colors = Record<'primary' | 'danger', string | [number, number, number]>

const annotated: Colors = { primary: '#06f', danger: [255, 0, 0] }
// annotated.primary.toUpperCase() // error: might be a tuple

const theme = {
  primary: '#06f',
  danger: [255, 0, 0],
} satisfies Colors

theme.primary.toUpperCase() // OK: still string
theme.danger.map((c) => c / 255) // OK: still a tuple
```

16

What are mapped types? How would you implement `Partial` or `Readonly`?

Hard



#advanced types #mapped types

A mapped type builds a new object type by **looping over the keys** of another: `{ [K in keyof T]: ... }`.

Inside you can:

- change each property's type (`T[K]` → `Promise<T[K]>`)

- add or remove modifiers: `?`, `readonly`, and remove them with `-?` and `-readonly`
- rename keys with `as`, often with template literal types (`get${Capitalize<K>}`)
- filter keys out by remapping them to `never`

Most built-in utility types (`Partial`, `Required`, `Readonly`, `Record`, `Pick`) are just mapped types.

Example

```
type MyPartial<T> = { [K in keyof T]?: T[K] }
type MyReadonly<T> = { readonly [K in keyof T]: T[K] }
type Mutable<T> = { -readonly [K in keyof T]: T[K] }
type Getters<T> = { [K in keyof T as `get${Capitalize<string & K>}`]: () => T[K] }

type User = { name: string; age: number }
const draft: MyPartial<User> = { name: 'Asha' }
const getters: Getters<User> = { getName: () => 'Asha', getAge: () => 30 }
console.log(draft, getters.getName(), getters.getAge())
```

17

What are conditional types?

Hard



#advanced types #conditional types

A conditional type picks a type based on a test, like a ternary at the type level: `T extends U ? X : Y`.

When `T` is a union, the condition is applied to each member separately (a **distributive** conditional type). That is how `Exclude` works: `type Exclude<T, U> = T extends U ? never : T` removes members, since unions drop `never`.

Wrap both sides in brackets (`[T] extends [U]`) to **turn distribution off**.

Conditional types power library typings: `NonNullable`, `ReturnType`, `Awaited`, and API helpers whose return type depends on the arguments. Combined with `infer`, they can extract types from other types.

Example

```
type IsString<T> = T extends string ? 'yes' : 'no'
type A = IsString<'hi'> // 'yes'
type B = IsString<42> // 'no'

type MyExclude<T, U> = T extends U ? never : T
type Status = 'idle' | 'loading' | 'error'
type Busy = MyExclude<Status, 'idle'> // 'loading' | 'error'

const a: A = 'yes'
const b: Busy = 'loading'
console.log(a, b)
```

18

What does the `infer` keyword do?

Hard



#advanced types #conditional types

`infer` declares a type variable inside the `extends` clause of a conditional type, which TypeScript **fills in by pattern matching**. It lets you extract a piece of another type.

Examples:

- `T extends Promise<infer V> ? V : T`: the value inside a promise
- `T extends (...args: any[]) => infer R ? R : never`: a function's return type (this is `ReturnType`)
- `T extends (infer E)[] ? E : never`: an array's element type
- template literal types, to parse strings such as route paths

You can only use `infer` in the condition part of a conditional type.

Example

```
type UnwrapPromise<T> = T extends Promise<infer V> ? V : T
type ElementOf<T> = T extends (infer E)[] ? E : never
type MyReturnType<F> = F extends (...args: any[]) => infer R ? R : never

async function loadUser() { return { id: 1, name: 'Asha' } }

type User = UnwrapPromise<MyReturnType<typeof loadUser>> // { id:
number; name: string }
type N = ElementOf<number[]> // number

const u: User = { id: 2, name: 'Ben' }
const n: N = 7
console.log(u.name, n)
```

19

What are declaration files (`.d.ts`)? How do you type a library that has no types?

Medium



#declarations #tooling

A `.d.ts` file contains **only type information**, no implementation. It describes the shape of JavaScript code so TypeScript can check how you use it. Libraries written in TypeScript generate them on build (`declaration: true`).

For an untyped package:

1. check for community types: `npm i -D @types/package-name` (DefinitelyTyped)
2. otherwise write your own: `declare module 'package-name' { ... }` in a `.d.ts` file in your project
3. as a quick escape hatch, `declare module 'package-name'` with no body types it as `any`

`.d.ts` files are also used to augment globals (`declare global { interface Window { ... } }`) and to type non-code imports like `*.svg` or `*.css`.

Example

```
// types/legacy-lib.d.ts
declare module 'legacy-lib' {
  export function slugify(input: string, sep?: string): string
}

// types/global.d.ts
declare global {
  interface Window { analytics?: { track(event: string): void } }
}
declare module '*.svg' {
  const src: string
  export default src
}
export {}
```

20

What are function overloads, and when would you use them?

Medium 🔥 #functions

Overloads let one function have **several call signatures**, so the return type can depend on which arguments are passed. You write the overload signatures first, then a single implementation whose signature is compatible with all of them (and is **not visible to callers**).

Example: `createElement('a')` returns an anchor type, `createElement('div')` a div type.

Often a simpler tool is better:

- a **union** parameter if the return type does not change
- a **generic** with a conditional or indexed return type when it does

Overloads are checked in order, so put the most specific one first.

Example

```
function parse(input: string): number
function parse(input: string[]): number[]
function parse(input: string | string[]): number | number[] {
  return Array.isArray(input) ? input.map(Number) : Number(input)
}

const one = parse('42')           // number
const many = parse(['1', '2'])    // number[]
console.log(one + 1, many.length)
```

21

Abstract class vs interface: what is the difference?

Medium 🔥🔥 #oop #classes

An **interface** is a pure contract: it has no implementation and disappears at compile time. A class can **implements** many interfaces.

An **abstract class** is a real class that cannot be instantiated directly. It can contain **shared implementation** and state, constructors, and **protected** members, plus **abstract** methods that subclasses must implement. A class can extend only one class.

Use an interface to describe a shape or capability (**Serializable**, **Repository**). Use an abstract class when subclasses genuinely share code, like a template method. Since an abstract class exists at runtime, **instanceof** works with it but not with an interface.

Example

```
interface Shape { area(): number }

abstract class Base implements Shape {
  constructor(protected name: string) {}
  abstract area(): number
  describe() { return `${this.name}: ${this.area().toFixed(1)}' } //
shared code
}

class Circle extends Base {
  constructor(private r: number) { super('circle') }
  area() { return Math.PI * this.r ** 2 }
}

// new Base('x') // error: cannot create an instance of an
abstract class
const c = new Circle(2)
console.log(c.describe(), c instanceof Base)
```

22

What is the difference between `private` and `#private` fields?

Easy

🔥👉 #oop #classes

- `public` (default), `protected` (class and subclasses) and `private` (class only) are **TypeScript modifiers**: they are checked at compile time and then erased. At runtime the property is an ordinary, accessible property, and `(obj as any).secret` works.
- `#secret` is a **JavaScript private field**: enforced by the runtime, truly inaccessible from outside, even with `any` or in the console.

TypeScript also offers `readonly` and **parameter properties** (`constructor(private repo: Repo)`), a shorthand that declares and assigns a field in one go.

Use `#` when you need real encapsulation (libraries); `private` is fine and common in application code.

Example

```
class Account {
  private pin = 1234          // compile-time only
  #balance = 100              // real runtime privacy
  get balance() { return this.#balance }
}

const acc = new Account()
console.log((acc as any).pin)  // 1234: private is erased
console.log((acc as any).balance) // 100 via the getter
console.log(Object.keys(acc))  // ['pin']: #balance is not a
                                property
```

23

How do you type data that comes from an API?

Medium 🔥🔥 #validation #api

Types are erased, so annotating `const user: User = await res.json()` is just a promise to the compiler: if the API returns something else, nothing catches it.

Options, from weakest to strongest:

- type the response as `unknown` and narrow it with type guards
- generate types from the API contract (OpenAPI, GraphQL codegen, tRPC sharing types end to end)
- validate at runtime with a schema library such as Zod or Valibot, and infer the TypeScript type from the schema (`z.infer<typeof UserSchema>`), so there is one source of truth

Validate at the boundary (API responses, form input, env variables), then trust the types inside your app.

Example

```
type User = { id: number; name: string }

function isUser(v: unknown): v is User {
  return typeof v === 'object' && v !== null
    && typeof (v as User).id === 'number'
    && typeof (v as User).name === 'string'
}

const good: unknown = JSON.parse('{ "id":1, "name": "Asha" }')
const bad: unknown = JSON.parse('{ "id": "1" }')
console.log(isUser(good) ? good.name : 'invalid')
console.log(isUser(bad) ? bad.name : 'invalid')
```

24

What are template literal types?

Hard



#advanced types

Template literal types build string types with the same syntax as JavaScript template strings, for example a type written as `on${Capitalize<Event>}` inside backticks.

When the placeholders are unions, TypeScript produces every combination: ``${'sm' | 'lg'}-${'red' | 'blue'}`` gives four string literal types.

Uses:

- typed event names, CSS class names or design tokens
- route params, like extracting `'id'` from `'/users/:id'` with `infer`
- key remapping in mapped types (`get${Capitalize<K>}`)

The intrinsic helpers `Uppercase`, `Lowercase`, `Capitalize` and `Uncapitalize` transform them.

Example

```
type Size = 'sm' | 'lg'
type Color = 'red' | 'blue'
type ButtonClass = `btn-${Size}-${Color}` // 4 combinations

type Params<P> = P extends `${string}:${infer Name}/${infer Rest}`
  ? Name | Params<Rest>
  : P extends `${string}:${infer Name}` ? Name : never

type R = Params<'/users/:userId/posts/:postId'> // 'userId' | 'postId'

const cls: ButtonClass = 'btn-lg-red'
const key: R = 'postId'
console.log(cls, key)
```

25

Which `tsconfig.json` options do you consider important?

Medium



#config #tooling

- `strict: true`: the essential one
- `noUncheckedIndexedAccess`: `arr[i]` and `record[key]` become `T | undefined`, catching out-of-bounds bugs
- `target` and `lib`: which JavaScript version to emit and which built-in APIs exist (`DOM`, `ES2022`)
- `module` and `moduleResolution`: how imports are resolved (`bundler` for Vite and Next.js, `nodenext` for Node)
- `paths` / `baseUrl`: import aliases like `@/components`
- `noEmit`: type-check only, when a bundler (esbuild, SWC) does the actual compiling
- `isolatedModules` / `verbatimModuleSyntax`: keep files safe for single-file transpilers
- `skipLibCheck`: skip checking `.d.ts` in `node_modules` for speed

Interviewers mostly want to hear `strict` and that you understand bundlers often only strip types.