

TypeScript Interview Questions and Answers

Intern to senior level

TypeScript interview questions and answers from beginner to senior: interface vs type, any vs unknown, generics, union types, utility types, type narrowing, enums vs unions, mapped types and inference.

10 questions Read online: <https://www.freedevtemplates.com/interview-questions/typescript>

1. What is TypeScript and why do we use it?

Level: intern TypeScript

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript. It catches type errors in the editor and at build time, and it documents the shape of data for the next person who reads the code.

Every valid JavaScript file is already TypeScript, which is why teams can adopt it gradually. The compiler (tsc) erases types and emits JS that runs anywhere JavaScript runs. Types exist only at compile time: they are not available for runtime checks unless you add a library or generate them.

The usual wins are safer refactors, better editor autocomplete, and APIs that are harder to call wrongly. The cost is a build step and time spent modelling awkward JavaScript patterns. Interviews often follow this with "when would you still use JavaScript?" - small scripts, prototyping, or a codebase that is not yet worth converting.

2. What is the difference between interface and type in TypeScript?

Level: intern TypeScript

Both describe object shapes. interface can be merged and is preferred for public object contracts; type can express unions, intersections and primitives, and cannot be reopened later.

For a simple object, they look the same and teams pick one style. interface supports declaration merging: two interface User blocks become one, which is how many .d.ts files extend library types. type aliases are fixed once written.

Use type when you need a union (string | number), a mapped type, or a tuple. Use interface when you are describing an object that others might extend. You can implement either with a class. Do not waste interview time claiming one is "faster"; they compile to the same nothing.

Key points

interface: extendable, mergeable, object-shaped

type: unions, intersections, primitives, tuples

Pick one style for objects and stay consistent

```
interface User { id: string; name: string }
interface User { email?: string } // merged

type Id = string | number; // union: type only
type Admin = User & { role: "admin" }; // intersection
```

3. What is the difference between any, unknown and never in TypeScript?

Level: junior TypeScript

any turns off checking. unknown is a type-safe any: you must narrow it before use. never means a value cannot exist, used for exhaustive checks and functions that always throw.

A value typed `any` can be passed anywhere and you can read any property on it, so a `typo` compiles. `unknown` is what you give to untrusted input (`JSON.parse`, a network payload) until you prove its shape with `typeof`, `in`, or a type guard.

`never` is the empty set. A function with return type `never` does not return (throw or infinite loop). In a `switch`, assigning the leftover value to `const _exhaustive: never = value` makes the compiler fail when a new union member is added and not handled.

```
function handle(x: unknown) {
  if (typeof x === "string") return x.toUpperCase();
  throw new Error("expected string");
}

function fail(message: string): never {
  throw new Error(message);
}
```

4. What are generics in TypeScript?

Level: junior TypeScript

Generics are type parameters: they let a function, class or type work with many types while keeping the relationship between inputs and outputs, instead of falling back to `any`.

`identity<T>(value: T): T` returns the same type it received. Without `T` you would write `any` and lose that link. Constraints (`T extends { id: string }`) describe the minimum shape. Several parameters (`K`, `V`) model maps and API wrappers.

Common interview follow-ups: default parameters (`T = string`), `keyof T` for safe property access, and why `Array<T>` and `T[]` are the same. Prefer inference (`wrap(user)`) over forcing `wrap<User>(user)` at every call site.

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

first(["a", "b"]); // string | undefined
first<number>([1, 2, 3]); // number | undefined
```

5. What are union and intersection types in TypeScript?

Level: junior TypeScript

A union (`A | B`) is a value that is one of the members. An intersection (`A & B`) is a value that must satisfy both types at once.

Unions model "this or that": a result that is `{ ok: true, data: T } | { ok: false, error: string }`. Narrow with a discriminant field (`ok`) before you read `data`. Intersections model "this and that": `User & { role: "admin" }` has every `User` field plus `role`.

A union of objects is not an object with optional fields. `A | B` only lets you access properties common to both until you narrow. That is the usual bug when someone writes `user.role` on a `User | Admin` without checking.

6. What are utility types in TypeScript like Partial, Pick and Omit?

Level: junior TypeScript

Utility types are built-in helpers that transform an existing type: `Partial` makes every field optional, `Required` the opposite, `Pick` keeps a set of keys, `Omit` removes them, and `Record` builds a key/value map.

`Partial<User>` is useful for a PATCH body. `Pick<User, "id" | "name">` is a list-row type. `Omit<User, "password">` is

what you send to the client. `Record<Id, User>` is a dictionary. `ReturnType<typeof fn>` and `Parameters<typeof fn>` pull types from functions so you do not repeat them.

You can write the same things with mapped types; the utilities are the readable version. Interviews sometimes ask you to implement `Partial` yourself: `{ [K in keyof T]?: T[K] }`.

```
type User = { id: string; name: string; password: string };
type PublicUser = Omit<User, "password">;
type UserPatch = Partial<Pick<User, "name" | "password">>;
```

7. What is type narrowing in TypeScript?

Level: mid TypeScript

Narrowing is when TypeScript shrinks a wide type (such as a union) to a more specific type after a runtime check like `typeof`, `instanceof`, `in`, or a custom type guard.

Control-flow analysis follows your `if` and `switch`. `typeof x === "string"` narrows `string | number` to `string` in that branch. `x instanceof Date` works for classes. `"width"` in `value` distinguishes object shapes. A user-defined guard is a function whose return type is `value` is `Fish`.

Equality checks against a discriminant (`event.type === "click"`) are the cleanest pattern for API payloads.

Assertions (`value as User`) skip narrowing and are a common source of production bugs: the compiler believes you, the runtime does not.

```
function isUser(value: unknown): value is { id: string } {
  return typeof value === "object" && value !== null && "id" in value;
}

function label(value: string | number) {
  return typeof value === "string" ? value.toUpperCase() : value.toFixed(2);
}
```

8. What is the difference between enum and union types in TypeScript?

Level: mid TypeScript

A string union such as `"admin" | "user"` exists only in the type system and erases to plain strings. A TypeScript enum compiles to a JavaScript object (unless it is a `const` enum), which is extra runtime code and a weaker default.

Most codebases prefer union types plus `as const` objects when they also need a runtime list (`const Roles = ["admin", "user"] as const`). Numeric enums are especially surprising because they reverse-map (`Color[0] === "Red"`). `const` enum inlines values but is banned in some projects because it breaks isolated modules and bundlers.

If a library already exposes an enum, use it. When you design a new API, a union is simpler to tree-shake, serialise and explain.

9. How does TypeScript type inference work?

Level: mid TypeScript

TypeScript fills in types you omit: literals get their literal or widened type, function return types are inferred from return statements, and generic arguments are inferred from the values you pass.

`let count = 0` becomes `number`. `const role = "admin"` stays the literal `"admin"` because `const` cannot be reassigned. Contextual typing works the other way: a callback argument is inferred from the function you pass it to (`(arr.map(item => ...))`).

Write types at module boundaries (function arguments, exported APIs, public class fields) and let inference handle locals. Over-annotating locals makes refactors noisy. When inference widens too far (`[]` is `any[]` without a hint), give the generic or a typed empty value (`[]` as `User[]` or `User[] = []`).

10. What are mapped types and conditional types in TypeScript?

Level: senior TypeScript

Mapped types build a new object type by looping over keys (`{ [K in keyof T]: ... }`). Conditional types pick result with `T extends U ? X : Y`, which is how many utility types are implemented.

`Partial<T>` is a mapped type that adds `?` to every key. You can also remap keys with `as` (`[K in keyof T as ${K & string}Id]`). Conditional types power `Exclude`, `Extract` and `NonNullable`. They become interesting with `infer`, which pulls a piece out of a type (`T extends Promise<infer U> ? U : T`).

Senior interviews look for judgement: these features belong in shared utility types, not in every component file. If a conditional type is hard to read, a few well-named helpers beat one clever one-liner. Recursion has a depth limit; it is the wrong tool for modelling arbitrarily nested JSON unless you must.

```
type ReadonlyFields<T> = { readonly [K in keyof T]: T[K] };
type Awaited<T> = T extends Promise<infer U> ? Awaited<U> : T;
```

These answers are written for real interview conversations. Prefer explaining a trade-off out loud over reciting a definition. Full explanations and related questions: <https://www.fredevtemplates.com/interview-questions/typescript>