## Chapter 15 Recap: Conditional Types — From Scratch to Mastery

### 🔹 1. Basic Concept

## 🔹 1. What is a Type?

A **type** defines what kind of value a variable can hold.

ts

CopyEdit

let name: string = "Likan"; // 'string' is the type

let age: number = 30; // 'number' is the type

## 🔹 2. What is a Custom Type Rule?

A **custom type rule** is a **type you create yourself** using type and <T> to apply some logic to types.

Syntax –

type YourRuleName<T> = /\* some logic using T \*/;

Basic syntax - type YourRuleName<T> = T

type YourRuleName → This is a \*\*type alias\*\*

type YourRuleName<T> → This is a \*\*generic type alias\*\*

## What is a **Type Alias** in TypeScript?

A **type alias** is a way to **give a name to a type** so you can reuse it.

// Instead of repeating this:

let username: string;

// You can do this:

type Username = string;

let username: Username;

type MyType = string;

Now MyType is just another name for string.

## **Generic Type Alias** in TypeScript?

### 🔹 Simple One-Liner:

A **generic type alias** is a type alias that works with **any type**, using a **type variable like <T>**.

It’s like a **type template** — you give it a type, and it returns a type (maybe the same or a modified one).

Syntax –

type MyGeneric<T> = T;

Because of <T> — it's **not fixed to one type**, it can accept any type dynamically.

Example –

// 1️⃣ Identity Type — returns the same type

type Identity<T> = T;

const userName: Identity<string> = "Likan";

const userAge: Identity<number> = 30;

// 2️⃣ Wrapper Type — wraps a type inside an object

type Wrapper<T> = { value: T };

const wrappedName: Wrapper<string> = { value: "Likan" };

const wrappedScore: Wrapper<number> = { value: 99 };

// 3️⃣ Nullable Type — allows value or null

type Nullable<T> = T | null;

const city: Nullable<string> = "Delhi";

const country: Nullable<string> = null;

// 4️⃣ Array Type — array of generic items

type MyArray<T> = T[];

const scores: MyArray<number> = [80, 90];

const names: MyArray<string> = ["A", "B"];

// 5️⃣ Function Type — generic function signature

type Callback<T> = (value: T) => void;

const printName: Callback<string> = (value:string) => console.log(value);

// 6️⃣ Conditional Type — filters non-string types

type OnlyStrings<T> = T extends string ? T : never;

type A = OnlyStrings<string>; // string

type B = OnlyStrings<number>; // never

**// 7️⃣ Mapped Type — make all props optional**

// 🔹 Step 1: Define the base object type

type User = {

name: string;

age: number;

};

// 🔹 Step 2: Create a generic mapped type to make all keys optional

type MakeOptional<T> = {

[K in keyof T]?: T[K];

// 🔁 Loop through each key K in T (like "name", "age")

// ❓ Add `?` to make each property optional

// 🎯 Keep the original value type (T[K])

};

// 🔹 Step 3: Apply it to the User type

type OptionalUser = MakeOptional<User>;

/\*

🔍 Resulting type:

type OptionalUser = {

name?: string;

age?: number;

}

\*/

// 🔹 Step 4: Real usage

const optionalUser: MakeOptional<User> = {

name: "Likan" // ✅ Only name provided, age is optional

// age can be omitted ✅

};

// 8️⃣ Intersection Type — merge two types

type WithId<T> = T & { id: string };

const itemWithId: WithId<{ title: string }> = {

title: "Book",

id: "abc123",

};

// 9️⃣ Union Type — add generic with extra options

type OrNullOrUndefined<T> = T | null | undefined;

const email: OrNullOrUndefined<string> = undefined;

## 🔹 3. What is a Conditional Type?

### In TypeScript, a **conditional type** allows you to create a **new type** depending on whether a **type condition is true or false**.

### ✅ Syntax:

type SomeType<T> = T extends ConditionType(U)

? TrueType (x)

: FalseType;(Y)

T extends ConditionType(U)

? TrueType (x)

: FalseType;(Y)

If T = ConditionType(U) return TrueType (x) else return FalseType;(Y)

### 📌 Example:

ts

CopyEdit

type IsString<T> = T extends string ? "yes" : "no";

This means:

* If T is a string → return "yes"
* If not → return "no"

### 🔧 Real Example:

ts

CopyEdit

type IsNumber<T> = T extends number ? "✅ number" : "❌ not a number";

ts

CopyEdit

type A = IsString<"hello">; // "✅ string"

type B = IsString<123>; // "❌ not string"

You can **nest** them too: type Check<T> =

T extends string ? "string" :

T extends number ? "number" :

T extends boolean ? "boolean" :

"unknown";

type A = Check<string>; // "string"

type B = Check<number>; // "number"

type C = Check<true>; // "boolean"

type D = Check<{}>; // "unknown"

**Usage -**

### 🔹 3. Real-World Use Case – Filtering Union Types

// Custom version of Exclude utility

type MyExclude<T, U> = T extends U ? never : T;

#### 📖 Explanation:

* For each type T in the union ("a", "b", "c"),
* If it **extends U** ("b"), we return never (remove it)
* Otherwise, we keep the type

type Result = MyExclude<"a" | "b" | "c", "b">;

final result -   
type Result = "a" | "c";

"b" is **excluded** from the union!

### 🧪 Real-World Scenario:

You have user roles:

ts

CopyEdit

type Role = "admin" | "editor" | "viewer";

You want to remove "admin" from access list:

ts

CopyEdit

type PublicRoles = MyExclude<Role, "admin">;

// "editor" | "viewer"

Now only non-admins get access ✅

Behind the scneeas -   
TypeScript **distributes** conditional types over unions.

T is: "a" | "b" | "c" — a union of **3 types**

So TypeScript breaks it into 3 separate checks:

php

CopyEdit

"a" extends "b" ? never : "a" |

"b" extends "b" ? never : "b" |

"c" extends "b" ? never : "c"

➡️ Final result:

ts

CopyEdit

"a" | never | "c" → "a" | "c"

Each member of the union is checked **individually**.

### 🔹 4. Built-in Utilities Based on Conditional Types

| **🔧 Utility** | **✅ Based on conditional types** |
| --- | --- |
| Exclude<T, U> | Keep everything in T **not** assignable to U |
| Extract<T, U> | Keep everything in T **that is** assignable to U |
| NonNullable<T> | Remove null and undefined |
| ReturnType<T> | Extract return type of a function |
| InstanceType<T> | Get instance type from constructor |
| Awaited<T> | Unwrap Promise like types |

### 🔹

### Use Cases of conditional types –

**✅ 1️⃣ Type Check — IsString<T>**

// ✅ Conditional Type: Check if a type is a string

type IsString<T> = T extends string ? "yes" : "no";

// ✅ Test the conditional type

type A = IsString<string>; // "yes"

type B = IsString<number>; // "no"

// ✅ Generic function using the conditional type

function checkType<T>(val: T): IsString<T> {

// At runtime, we check the value type

// At compile-time, we assert it as IsString<T>

return (typeof val === "string" ? "yes" : "no") as IsString<T>;

}

// ✅ Using the function

const result1 = checkType("hello"); // "yes"

const result2 = checkType(123); // "no"

Is as IsString<T> **mandatory** in this code?

Yes , unless TypeScript will throw an error

 if we write return return (typeof val === "string" ? "yes" : "no"); , The expression (typeof val === "string" ? "yes" : "no") returns "yes" | "no" (a union)

 But the function return type IsString<T> expects **either "yes" or "no"**, based on T

 ❌ **"TypeScript can't safely assign a union ("yes" | "no") to a specific literal type ("yes" or "no")."**

✅ This is absolutely true — TypeScript won't allow you to assign a **wider** type (like a union) to a **narrower** type (like a specific literal).

✅ Proof: TypeScript won't assign a union to a specific literal type -

type OnlyYes = "yes"; // A specific literal type

let union: "yes" | "no" = "yes"; // This is a union type

let exact: OnlyYes = union; // ❌ TypeScript will throw an error

❌ TypeScript Error:

Type '"yes" | "no"' is not assignable to type '"yes"'.

Type '"no"' is not assignable to type '"yes"'.

### ✅ Why?

Even though union currently holds the value "yes", its **type** is "yes" | "no",  
so TypeScript is protecting you by saying:

“Yo, I can't be sure this won’t be "no" later, so I won’t allow you to assign it to a variable that only accepts "yes".”

This is TypeScript's **strict safety rule**:

❌ A **wider type (union)** cannot be assigned to a **narrower type (literal)**

## 🧠 Bonus Example (for "no"):

ts

CopyEdit

type OnlyNo = "no";

let result: "yes" | "no" = "no";

let confirmed: OnlyNo = result; // ✅ Works only if value is "no"

## ✅ Fix (only if you're sure):

If you really want to force it, you have to manually **assert the type**:

ts

CopyEdit

let exact: OnlyYes = union as OnlyYes; // ✅ Now allowed, but you take responsibility

## ❌ What happens without as IsString<T>?

ts

CopyEdit

function checkType<T>(val: T): IsString<T> {

return typeof val === "string" ? "yes" : "no"; // ❌ Error

}

### ❗ Error:

Type '"yes" | "no"' is not assignable to type 'IsString<T>' which is a literal type.

Because TS thinks:

* The expression might return both "yes" and "no"
* But IsString<T> could be just "yes" or just "no" depending on T

So it **cannot confirm it's safe**.

The return value must exactly match the result of the conditional type (IsString<T>) — either "yes" or "no", not the union "yes" | "no". That’s why we use as IsString<T> to assert it manually.

Can a **narrower type** (like "yes") be assigned to a **wider type** (like "yes" | "no")?”

### ✅ ****Yes — the vice versa is allowed.****

That means:

ts

CopyEdit

let narrow: "yes" = "yes"; // ✅ a string literal type

let wide: "yes" | "no" = narrow; // ✅ ✅ valid

✅ This works because:

A "yes" **is definitely** a valid member of "yes" | "no".  
You're assigning a **narrower value to a wider type**, which is always safe.

**✅ 2️⃣ UI Field Type Based on Input Type**

// ✅ Conditional type: returns a string literal type based on T

type FieldType<T> = T extends string ? "TextInput" : "NumberInput";

// ✅ Generic function that renders an input based on type

function renderField<T>(value: T, fieldType: FieldType<T>) {

if (fieldType === "TextInput") {

console.log("Render <input type='text'>"); // ✅ string

} else {

console.log("Render <input type='number'>"); // ✅ number

}

}

// ✅ Usage examples — TS checks the correct fieldType for given value

renderField("name", "TextInput"); // OK: value is string

renderField(99, "NumberInput"); // OK: value is number

FieldType<T> uses a **conditional type** to infer "TextInput" if T is a string, else "NumberInput". This ensures **type-safe rendering logic** tied directly to the data type.

**✅ 3️⃣ Filter Union Types — Custom Exclude<T, U>**

type MyExclude<T, U> = T extends U ? never : T;

// ✅ Remove types from T that match U by returning `never`

type Role = "admin" | "editor" | "viewer";

type PublicRoles = MyExclude<Role, "admin">; // "editor" | "viewer"

function access(role: PublicRoles) {

console.log(`${role} is allowed.`);

}

access("editor"); // ✅ allowed

// access("admin"); // ❌ Error, removed by MyExclude

**✅ 4️⃣ API Response Shape Based on Status**

// ✅ 4️⃣ API Response Shape Based on Status using Conditional Types

// 👇 This conditional type checks the value of T and returns a different type accordingly

type APIResponse<T> =

T extends "success" ? { data: string } : // If T is "success" → return this shape

T extends "error" ? { error: string } : // If T is "error" → return this shape

never; // If T is anything else → error (never)

// 👇 Generic function that uses the conditional type above

function handleResponse<T extends "success" | "error">(status: T, res: APIResponse<T>) {

if (status === "success") {

console.log(res.data);

} else {

console.log(res.error);

}

}

// ✅ Case 1: T is "success" → APIResponse<T> = { data: string }

handleResponse("success", { data: "OK" }); // ✅ Works fine

// ✅ Case 2: T is "error" → APIResponse<T> = { error: string }

handleResponse("error", { error: "Something went wrong" }); // ✅ Works fine

// ❌ Case 3: T is "success" → APIResponse<T> = { data: string }

// But you gave wrong shape { error: "..." } → Compile-time error

// handleResponse("success", { error: "fail" }); // ❌ Error

// ❌ Case 4: T is "error" → APIResponse<T> = { error: string }

// But you gave wrong shape { data: "..." }

// handleResponse("error", { data: "fail" }); // ❌ Error

### 🔷 <T extends "success" | "error"> — Why It’s Needed (With Proof)

ts

CopyEdit

type APIResponse<T> =

T extends "success" ? { data: string } :

T extends "error" ? { error: string } :

never;

function handleResponse<T extends "success" | "error">(status: T, res: APIResponse<T>) {

// ✅ 1. Safe narrowing — TS knows res shape based on status

if (status === "success") {

console.log(res.data); // ✅ res = { data: string }

} else {

console.log(res.error); // ✅ res = { error: string }

}

}

// ✅ 2. Correct return type from APIResponse<T>

handleResponse("success", { data: "ok" }); // ✅ APIResponse<"success"> = { data: string }

handleResponse("error", { error: "fail" }); // ✅ APIResponse<"error"> = { error: string }

// ❌ 3. Prevents wrong combinations

// handleResponse("loading", { data: "wait" }); // ❌ "loading" not allowed — T must extend "success" | "error"

// handleResponse("success", { error: "fail" }); // ❌ APIResponse<"success"> = { data }, but 'erro

## **🔶 Case 2: T (No Restriction ❌ — Open & Unsafe)**

ts

CopyEdit

type APIResponse<T> =

T extends "success" ? { data: string } :

T extends "error" ? { error: string } :

never;

function handleResponse**<T>(**status: T, res: APIResponse<T>) {

// ❌ No narrowing — TS doesn’t know what T is

// console.log(res.data); // ⚠️ Error: Property 'data' does not exist (could be anything)

// console.log(res.error); // ⚠️ Same issue here

}

handleResponse("success", { data: "ok" }); // ⚠️ Looks fine, but no strict validation on ttype T . Code works at runtime and compiles fine

handleResponse("error", { error: "fail" }); // ⚠️ Same — TS guesses, not guarantees.no strict validation on tytpe T . Code works at runtime and compiles fine

handleResponse("loading", { data: "wait" });

// ❌ No restriction — TS \*\*initially allows\*\* "loading" as T, but

// APIResponse<"loading"> evaluates to `never`, so res: { data } causes a ✅ compile-time error.

// 👉 To fix this: we need `T extends "success" | "error"` to restrict T upfront and avoid this failure.

// T is "loading"

// APIResponse<"loading"> → **never**

handleResponse("success", { error: "fail" }); // ❌ TS can't catch mismatch properly. TypeScript gives a compile-time error here

//  T = "success" → APIResponse<"success"> = { data: string }

// You’re passing { error: "fail" } instead of { data: string }

**✅ 5️⃣ Remove Null & Undefined — Custom NotNull<T>**

// ✅ 5️⃣ Remove Null & Undefined — Custom NotNull<T>

// 👇 This conditional type checks:

// If T is null or undefined → return never (which causes a compile-time error)

// Else → return the original type T

type NotNull<T> = T extends null | undefined ? never : T;

// 👇 This function only accepts values that are \*not\* null or undefined

function safeAssign<T>(value: NotNull<T>) {

console.log("Assigned:", value); // We are now guaranteed that 'value' is valid

}

// ✅ This works because "Likan" is a valid string (not null/undefined)

safeAssign("Likan");

// ❌ These would give compile-time errors:

// safeAssign(null); // Error: Argument of type 'null' is not assignable

// safeAssign(undefined); // Error: Argument of type 'undefined' is not assignable

NotNull<T> helps **eliminate null or undefined** from a type, giving you **compile-time protection** for required values.

**✅ 7️⃣ Prop Shaping Based on Variant (React-style)**

// ✅ Conditional type: decides prop shape based on variant

type ButtonProps<T> =

T extends "link" ? { href: string } :

T extends "primary" ? { onClick: () => void } :

never;

// ✅ Function with generic T, but T is inferred from props

function Button<T extends "link" | "primary">(props: ButtonProps<T>) {

if ("href" in props) {

console.log("It’s a link:", props.href);

} else {

props.onClick();

}

}

// ✅ TypeScript auto-detects the type from props (T is inferred)

Button({ href: "https://google.com" }); // ✅ T inferred as "link"

Button({ onClick: () => console.log("clicked") }); // ✅ T inferred as "primary"

// ❌ Wrong usages:

// Button({ error: true }); // ❌ No matching shape — T cannot be inferred

// Button({ href: "url", onClick: () => {} }); // ❌ Invalid: TS can’t decide if T is "link" or "primary"

### // Button<T> here pasing <T> during tehcalll is opotional because ts automatically infers it

### 5. infer Keyword

### The infer keyword in TypeScript is used inside a conditional type to extract a part of a type, give it a temporary name, and use that name in the true branch of the condition.

**“Extract a part of a type”** means — for example, if the type is a callback signature, infer can extract a specific part like its return type.

**“Give it a temporary name”** means — once that part is extracted, it’s temporarily assigned a name using infer R.

**“Use it later”** means — if the condition is true, we can use R in the ? branch of the conditional type.

✅ 1. Extract Return Type of a Function

// ✅ Define a generic type GetReturn<T>

// It checks if T is a function like (...args: any[]) => something

// If yes, it extracts the return type using infer R; otherwise, it returns never

type GetReturn<T> = T extends **(...args: any[]) => infer R** ? R : never;

// ✅ Example 1: Function with NO parameters

type A = GetReturn<() => string>; // ✅ A = string

// 🧠 How this works:

// T = () => string

// It matches (...args: any[]) → because [] (no args) is still valid for any[]

// ✅ In TypeScript, an empty array [] is a subtype of any[]

// So even if no arguments are passed, it still satisfies (...args: any[])

// Therefore, R = string gets extracted

✅ 2. Extract Parameter Types of a Function

**// ✅ 1️⃣ Get ALL PARAMETERS as a tuple**

**type GetParams<T> = T extends (...args: infer P) => any ? P : never;**

**function login(email: string, password: string, rememberMe: boolean) {}**

**type P1 = GetParams<typeof login>;**

**// ✅ P1 = [email: string, password: string, rememberMe: boolean]**

**// 🧠 infer P grabs the full parameter list of the function**

**// ✅ 2️⃣ Get FIRST PARAMETER only**

**type FirstArg<T> = T extends (arg: infer A, ...rest: any[]) => any ? A : never;**

**function sendEmail(to: string, subject: string) {}**

**type P2 = FirstArg<typeof sendEmail>;**

**// ✅ P2 = string**

**// 🧠 infer A grabs just the first argument from the function**

3. Extract Element Type from an Array

type ElementType<T> = T extends (infer E)[] ? E : never;

type A = ElementType<string[]>; // string

type B = ElementType<number[]>; // number

🧠 **What infer is doing here:**  
It checks if the type is an array. If yes, it **grabs the type of each element** and names it E.

### ✅ 4. Extract Type from a Promise

ts

CopyEdit

type UnwrapPromise<T> = T extends Promise<infer R> ? R : T;

type A = UnwrapPromise<Promise<number>>; // number

type B = UnwrapPromise<string>; // string

🧠 **What infer is doing here:**  
It checks if the type is a Promise. If yes, it **extracts what the promise resolves to** and calls it R.

### ✅ 6. Extract Value Type from a Record/Object

ts

CopyEdit

type ValueType<T> = T extends Record<string, infer V> ? V : never;

type A = ValueType<{ name: string; age: number }>;

// string | number

🧠 **What infer is doing here:**  
It checks if the type is an object with string keys. If so, it **extracts all the value types** and combines them as a union.

7.Eracting Keys Using infer K

// 🔷 Custom utility to extract key types using `infer K`

type ExtractKeys<T> = T extends Record<infer K, any> ? K : never;

// 🔷 Sample object type

type Person = {

name: string;

age: number;

isAdmin: boolean;

};

// ✅ Usage

type KeysFromInfer = ExtractKeys<Person>; // "name" | "age" | "isAdmin"

type KeysFromKeyof = keyof Person; // "name" | "age" | "isAdmin"

// ✅ Both give the same result: union of key names as string literal types

### ✅ 7. Extract Instance Type from a Class

ts

CopyEdit

class Dog {

bark() {}

}

type DogInstance = InstanceType<typeof Dog>; // Dog

🧠 **What infer is doing here (internally):**  
TypeScript uses infer internally in InstanceType<T> to **extract the type returned by new**, i.e., the instance type.

### ✅ 8. Extract Constructor Parameter Types from a Class

ts

CopyEdit

class User {

constructor(public name: string, public age: number) {}

}

type Params = ConstructorParameters<typeof User>;

// [string, number]

🧠 **What infer is doing here (internally):**  
TS uses infer to extract the **constructor args list** from the class.

### ✅ 9. Extract Type from Nested Generics

ts

CopyEdit

type DeepExtract<T> = T extends Promise<Promise<infer U>> ? U : never;

type A = DeepExtract<Promise<Promise<string>>>; // string

🧠 **What infer is doing here:**  
It unwraps one layer of promise inside another — digging deeper and extracting U from the inner one.

### ✅ 10. Extract Union of Keys or Props

ts

CopyEdit

type Keys<T> = T extends { [K in keyof T]: infer V } ? V : never;

type A = Keys<{ a: number; b: string }>; // number | string

🧠 **What infer is doing here:**  
It loops over all properties and uses infer to **grab each value**, then returns the union of those types.

{ [K in keyof T]: infer V }

🤔 What does this do?

This is a mapped type, where you're saying:

"Loop through all keys (K) in the object T... and for each key, give me its value type and call it V."

🧠 Think of it like a template:

If you had this object:

ts

Copy

Edit

type T = { a: number; b: string };

Then:

keyof T means it returns union of keys

[K in keyof T] means lopp through all union of keys

{ [K in keyof T]: infer V }

becomes:

ts

Copy

Edit

{

a: V; // where V = number

b: V; // where V = string

}

Now TypeScript tries to figure out what V is — and here's the magic:

It sees a: number, b: string

So it unifies both number and string into:

👉 V = number | string

## ✅ That’s why your final output becomes a union of all property types 🧠 Interview Q&A

### Q1: What is a conditional type?

A way to write **type-level if-else logic** using T extends U ? X : Y.

### Q2: Where are conditional types used in TS?

* Utility types like Exclude, Extract, ReturnType
* Function return inference with infer
* Complex conditional logic in generic libraries

### Q3: What is infer?

A TypeScript keyword that **captures** a type part (e.g., function return) and lets you reuse it.

### Q4: What happens if T extends U is false?

The conditional returns the else branch (after :)

### Q5: Can we return never in conditional types?

✅ Yes. Used to **eliminate** certain values/types. Like:

ts

CopyEdit

type ExcludeNull<T> = T extends null ? never : T;

## ✅ Summary Snapshot

| **Concept** | **One-liner Description** |
| --- | --- |
| Conditional Type | T extends U ? X : Y |
| never | Used to discard type branches |
| infer | Capture part of a type |
| Real-world use | Exclude, Extract, ReturnType, Unwrapping Promises |

Would you like a visual image or just proceed to ✅ Chapter 16: infer, contextual typing, and type inference deep dive?

Top of Form

Tools

Bottom of Form

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

## ✅ 1. let value: string = "hello"

### 📌 This is called ****type annotation****.

It means:

“I am **declaring a variable** and I want it to always hold a string.”

### 🧱 Breakdown:

* value is the variable name.
* : string is the type annotation.
* "hello" is the value you're assigning.

ts

CopyEdit

let value: string = "hello"; // ✅ value is a string

value = "world"; // ✅ allowed

value = 123; // ❌ Error! 123 is not a string

👉 You're telling TypeScript:

“The only allowed type for this variable is string.”

## ✅ 2. "hello" as string

### 📌 This is called ****type assertion****.

It means:

“I already have a value — now I want to **force** TypeScript to treat it as a string.”

### 🧱 Breakdown:

* "hello" is a literal value.
* as string tells TypeScript to pretend or assume it's a string.

ts

CopyEdit

let random: unknown = "hello";

let length = (random as string).length; // ✅ We are saying: “Trust me, this is a string”

👉 You're telling TypeScript:

“You may not know what this is, but **I do** — it’s a string.”

## ✅ When do we use it?

There are **4 real-world situations** where you’d use as:

### 🔹 1. When TypeScript doesn’t know the type (e.g., any or unknown)

ts

CopyEdit

let data: unknown = "hello";

// TS doesn’t let you access `.length` on unknown

let length = (data as string).length; // ✅ Works after asserting it's a string

### 🔹 2. When working with DOM / HTML elements

ts

CopyEdit

const input = document.querySelector("input") as HTMLInputElement;

input.value = "Likan"; // ✅ TS knows it's an input now

### 🔹 4. When using conditional types

ts

CopyEdit

type IsString<T> = T extends string ? "yes" : "no";

function checkType<T>(val: T): IsString<T> {

// TS doesn’t know the exact return value at runtime

return (typeof val === "string" ? "yes" : "no") as IsString<T>;

}

Here, we use as IsString<T> to match the expected return type of the function.

## ✅ Summary (Interview One-Liner):

as <Type> is used to **manually assert a type** when TypeScript can’t figure it out, or when you want to **override** its inference.

### 🔹 Forcing a Type Using as Can Be Dangerous Without Proper Checks

function printLength(val: string | number) {

console.log((val as string).length); // ⚠️ You’re forcing TS to treat val as string

}

⚠️ This is dangerous if val is a number — you’ll get a runtime error!

### 1️⃣ Compile-time declaration (normal usage)

ts

CopyEdit

let x: string = "hello";

* ✅ This tells TypeScript: **“x must be a string.”**
* If you try to assign a number later: ❌ TS will throw an error.
* 🧠 This is **compile-time checking** — TS enforces the type **before running the code**.

### 2️⃣ Type Assertion using as

ts

CopyEdit

let something: any = "hello";

let strLength = (something as string).length;

* ✅ This is a **type assertion** — you're telling TypeScript:

"I know this is a string — treat it like one."

* 🧠 Still happens at **compile time**, not at runtime.
* ❌ It does **not** convert types — it only tells the compiler how to treat it.

### ✅ Is <T> Required When Calling a Generic Function in TypeScript?

**No, specifying <T> during a function call is optional** in most cases.

🔹 TypeScript is usually smart enough to **infer the type** of T from the arguments you pass to the function.  
🔹 However, in rare situations, you **might want to provide it explicitly** — especially when inference fails or needs to be overridden.

### ✅ Example 1: ****Type Inference (Recommended Way)****

ts

CopyEdit

function identity<T>(value: T): T {

return value;

}

const result1 = identity("hello"); // ✅ T is inferred as string

const result2 = identity(123); // ✅ T is inferred as number

⏺ **You don’t need to write <string> or <number> manually.**

### ✅ Example 2: ****Explicit Generic (Optional But Valid)****

ts

CopyEdit

const result1 = identity<string>("hello");

const result2 = identity<number>(123);

⏺ Works the same as inference, but you’re **forcing the type manually.**

### ❗ Example 3: ****Why Explicit**** <T> ****Might Be Needed****

ts

CopyEdit

function parse<T>(input: string): T {

return JSON.parse(input);

}

const result = parse('{"name": "Likan"}');

// ❌ TypeScript infers T as unknown → can't access .name

const resultFixed = parse<{ name: string }>('{"name": "Likan"}');

// ✅ Now resultFixed.name is accessible

⏺ **Here, you must specify <T> explicitly**, or TypeScript won't know the shape of the return type.Top understand this lets understand the below key points .

📘 Inference vs Type Assertion When Calling a Generic Function in TypeScript

### ✅ Case 1: Type Inference (T is used in parameters)

ts

CopyEdit

function identity<T>(value: T): T {

return value;

}

const result = identity("hello"); // ✅ T is inferred as string

🧠 **Why this works:**

* The passed value "hello" matches the T in the parameter.
* TypeScript directly infers T = string.

### ❌ Case 2: No Inference Possible (T not used in parameter)

ts

CopyEdit

function parse<T>(input: string): T {

return JSON.parse(input);

}

const result = parse('{"name": "Likan"}');

// ❌ TS can't infer T — it's only in return type

// `result` becomes `any` or `unknown`

// because we don’t have T in fnction parameter ut cant infefr the arguments type. If T is **not used** in the **parameter list**, TypeScript has **no way to infer what T should be** — even if you're passing a valid argument that matches what T would be.

### ✅ Solution: Use Type Assertion via Explicit <T>

ts

CopyEdit

const result = parse<{ name: string }>('{"name": "Likan"}');

// ✅ TS now knows T = { name: string }

// You can safely access: result.name

🧠 **This is a type assertion via a generic argument**:  
You're telling TS: “I guarantee this will return type T.”  
Why did we use <T> in this call?

## Because T is used in the **return type** of the function

### Let's look at the function definition again:

ts

CopyEdit

function parse<T>(input: string): T {

return JSON.parse(input);

}

* This function says:  
  "I’ll take a string and return something of type T"
* But TypeScript doesn’t know **what T is** unless you:
  + Infer it from parameters ✅ (not possible here because input is just *string*)
  + **Or explicitly provide <T>** ❗

### 📌 Why you must pass <T> here:

ts

CopyEdit

parse<{ name: string }>('{"name": "Likan"}');

* You're saying:

“For this call, treat the return type (T) as { name: string }.”

* So now the function becomes (in TypeScript's eyes):

ts

CopyEdit

function parse(input: string): { name: string } {

return JSON.parse(input);

}

✅ This gives you **full type safety**:

ts

CopyEdit

result.name; // ✅ Allowed, type is known

## 🔥 Why this is Called a "Type Assertion via Generic Argument"

Because you're **asserting** the type of the return value by telling TypeScript what T is.

This is **different from as assertions**:

ts

CopyEdit

const result = JSON.parse(...) as { name: string }; // traditional cast

Using <T> with generics:

ts

CopyEdit

const result = parse<{ name: string }>('...');

✔️ More **reusable**, **type-safe**, and **maintainable** than as.

**Without passing <T> in the function call, the return type will be any (or unknown)**, and you'll **lose type safety**.

function parse<T>(input: string): T {

return JSON.parse(input);

}

const result = parse('{"name": "Likan"}');

// ⛔ T is not specified

// ⛔ TS can’t infer T from the input

// 🔁 So T becomes `any` (or `unknown`, depending on config)

Now result is of type any, so:

ts

Copy

Edit

result.name; // ✅ No error, but ❌ no IntelliSense or type safety

result.foo.bar.baz(); // ❌ Won’t catch errors

You’re basically back to using JavaScript-style dynamic typing — and this defeats the purpose of TypeScript.

✅

## ✅ With <T> — Return Type is Safe & Known

ts

CopyEdit

const result = parse<{ name: string }>('{"name": "Likan"}');

Now:

* TypeScript knows T = { name: string }
* So result is **typed**, and you get:

ts

CopyEdit

result.name; // ✅ string

result.age; // ❌ Error: age does not exist

result.toFixed(); // ❌ Error: not a number

## 🔍 Why TS Can't Infer T Here

Because the argument is just:

ts

CopyEdit

input: string

And T isn’t used in the parameter type. So TS says:

“You didn’t give me any clue for what T should be. So I’ll default to any (or unknown).”

✅ 1. What Does <T> Next to the Function Name Do?

When you write:

ts

CopyEdit

parse<{ name: string }>('{"name": "Likan"}');

You're telling TypeScript:

💬 “**For this call**, treat T as { name: string }.”

This is called an **explicit generic argument**.

So even if T is not in the parameter types, TS will **substitute T with { name: string }** **wherever it appears** (e.g., in the return type).

## ✅ 2. What Happens Inside the Function?

ts

CopyEdit

function parse<T>(input: string): T {

return JSON.parse(input); // 👈 JSON.parse returns `any`

}

* T is only used in the **return type**.
* So TS will treat this like:

function parse(input: string): { name: string } {

return JSON.parse(input); // 👈 Treated as returning that exact shape

}

Thanks to:

parse<{ name: string }>(...)

🧠 **Even though T isn't in the parameters, the return type is updated!**

## ❗

## So Why This Works

❗ TypeScript substitutes T **wherever it’s used** — **even if it’s only in the return type**.

As long as you pass <T> explicitly, **TS knows exactly what T is** — and can apply it.

## Key Rule

🔥 If TypeScript **can’t infer T**, you must **supply <T> manually**, and it will apply it **wherever T is used — even if only in the return type**.