**📘 Chapter 13: Mapped Types**

🔁 Dynamically transform and reshape types in TypeScript

I'll follow your preferred format:

**✅ 1. Concept Overview**

**Mapped types** allow you to create new types by transforming the properties of an existing type into a new type — this transformed type is called a mapped type. It works like looping through the keys of an object and modifying their types.

Make All Properties Optional

// ✅ Original type

type Person = {

name: string; // string property

age: number; // number property

};

// ✅ Mapped type to make all properties optional

type PartialPerson = {

// [K in keyof Person] means:

// Loop through each key K ("name" | "age") in the Person type

// For each key, assign its corresponding type Person[K] (string or number)

// The `?` makes each property optional

[K in keyof Person]?: Person[K]; // // ❓ Make all properties optional

};

// Resulting type:

// type PartialPerson = {

// name?: string;

// age?: number;

// }

✅ Mapped Type – Make Fields Nullable (No Generics)

// 🎯 Base type

type User = {

id: number;

name: string;

isActive: boolean;

};

// ✅ Mapped type directly for User (no <T>)

type NullableUser = {

[K in keyof User]: User[K] | null; // 👉 Add '| null' to each field manually

};

// ✅ Result:

// type NullableUser = {

// id: number | null;

// name: string | null;

// isActive: boolean | null;

// }

const user: NullableUser = {

id: null,

name: "Likan",

isActive: null

};

### ✅ 1. Make All Properties Readonly

ts

CopyEdit

type Person = { name: string; age: number };

type ReadonlyPerson = {

readonly [K in keyof Person]: Person[K]; // 🔒 Make all properties readonly

};

// Result: { readonly name: string; readonly age: number }

### ✅ 2. Change All Property Types to boolean

ts

CopyEdit

type Person = { name: string; age: number };

type BoolFlags = {

[K in keyof Person]: boolean; // override all value types to boolean

};

// Result: { name: boolean; age: boolean }

### ✅ 3. Add a Prefix to Keys (with template literals)

type Person = { name: string; age: number };

// // 🔁 Add prefix to all keys using `as`

type PrefixedKeys = {

[K in keyof Person as `person\_${K}`]: Person[K];

};

// Result: { person\_name: string; person\_age: number }

### ✅ 4. Remove null and undefined from each property

type Person = { name: string | null; age: number | undefined };

type NonNullablePerson = {

[K in keyof Person]: NonNullable<Person[K]>;

};

// Result: { name: string; age: number }

### ✅ 3. TypeScript: Conditional Type Transformation with Mapped Types

// 🔹 Original object type

type User = {

id: number;

name: string;

email: string;

isActive: boolean;

};

// 🔹 Conditional Transformation using mapped + conditional types

type Nullable<T> = {

// Loop through each key K in the object type T

[K in keyof T]:

// ✅ If the property's type is a string, transform it to string | null

T[K] extends string

? string | null

: T[K]; // ❌ Otherwise, keep the original type unchanged

};

// 🔹 Use Nullable<User> directly — no need for a separate alias

const user: Nullable<User> = {

id: 1, // number → unchanged

name: null, // ✅ was string → now string | null (null allowed)

email: "likan@example.com", // ✅ still valid — string allowed

isActive: true // boolean → unchanged

};

/\*

Final shape of Nullable<User>:

{

id: number; // ✅ stays number

name: string | null; // ✅ string → string | null

email: string | null; // ✅ string → string | null

isActive: boolean; // ✅ stays boolean

}

\*/

### ✅ 4. **Filtering Object Keys Using Conditional Remapping**

// 🔹 Original object type

type User = {

id: number;

name: string;

email: string;

isActive: boolean;

};

// 🔹 Utility: Filter to keep only keys whose values are strings

type StringKeysOnly<T> = {

[K in keyof T // 🔁 Loop over all keys of T

as T[K] extends string // 🧠 If the value type is string

? K // ✅ Keep the key

: never // ❌ Else drop the key

]: T[K]; // 🎯 Keep the original value type

};

// 🔹 Use it directly

type Filtered = StringKeysOnly<User>;

/\*

Final result:

type Filtered = {

name: string;

email: string;

}

\*/

// 🔹 Example usage

const userOnlyStrings: StringKeysOnly<User> = {

name: "Likan", // ✅ allowed — string

email: "likan@mail.com"// ✅ allowed — string

// id: 1 ❌ Error: 'id' was removed from the type

// isActive: true ❌ Error: 'isActive' was removed too

};

.

## 🧠 WHAT as DOES IN MAPPED TYPES

The as keyword in a mapped type **renames, filters, or transforms** keys **while mapping them**.

### 🔧 General Form:

ts

CopyEdit

type Mapped<T> = {

[K in keyof T as NEW\_KEY]: NEW\_VALUE;

};

## ✅ 1. **Filter Keys by Value Type**

ts

CopyEdit

type User = {

id: number;

name: string;

email: string;

active: boolean;

};

type StringKeysOnly<T> = {

[K in keyof T as T[K] extends string ? K : never]: T[K];

};

// Output:

type Result1 = StringKeysOnly<User>;

/\*

{

name: string;

email: string;

}

\*/

✅ 2. **Rename Keys with Prefix or Suffix**

type User = {

id: number;

name: string;

};

type PrefixedKeys<T> = {

[K in keyof T as `user\_${K}`]: T[K];

};

type Result = PrefixedKeys<User>;

/\*

{

user\_id: number;

user\_name: string;

}

\*/

K **or** string & K

string & K -> this checks all the keys must be string

✅ 3. **Change Case of Keys (Uppercase, Lowercase, etc.)**

type Settings = {

theme: string;

mode: string;

};

type UppercaseKeys<T> = {

[K in keyof T as Uppercase<string & K>]: T[K]; // ✅ using string & K

};

type Result3 = UppercaseKeys<Settings>;

/\*

{

THEME: string;

MODE: string;

}

\*/

**string & is compulsort this chekcs all the keys must be string type .**

## ✅ 4. **Remove Specific Keys**

ts

CopyEdit

type Employee = {

id: number;

name: string;

role: string;

};

type WithoutId<T> = {

[K in keyof T as K extends "id" ? never : K]: T[K];

};

// Output:

type Result4 = WithoutId<Employee>;

/\*

{

name: string;

role: string;

}

\*/

## ✅ 5. **Conditionally Transform Keys Based on Value Type**

ts

CopyEdit

type Product = {

title: string;

price: number;

inStock: boolean;

};

type NullableStrings<T> = {

[K in keyof T as T[K] extends string ? K : never]: string | null;

};

// Output:

type Result5 = NullableStrings<Product>;

/\*

{

title: string | null;

}

\*/

## ✅ 6. **Transform Both Keys & Values**

ts

CopyEdit

type FeatureFlags = {

login: boolean;

payment: boolean;

};

type FlagMap<T> = {

[K in keyof T as `has\_${string & K}`]: "enabled" | "disabled";

};

// Output:

type Result6 = FlagMap<FeatureFlags>;

/\*

{

has\_login: "enabled" | "disabled";

has\_payment: "enabled" | "disabled";

}

\*/

**✅ 2. Key Benefits**

// 🎯 Base type used across the app

type User = {

  id: number;

  name: string;

  isActive: boolean;

};

// ✅ Mapped type to make all properties optional

type OptionalUser = {

  [K in keyof User]?: User[K]; // 🔁1.1 DRY: Reuses keys and types from 'User' instead of repeating manually

};

/\*

❌ 1.2 Without mapped types, you'd write this manually:

type OptionalUserManual = {

  id?: number;

  name?: string;

  isActive?: boolean;

};

🧠 Imagine doing this for 50+ fields. Tedious & error-prone.

\*/

const updateUser: OptionalUser = {

  name: "Likan", // ✅ Only updating one field, thanks to optional mapping

};

/\*

  🛠 2 Safe Refactoring

  🚫 You should NOT redeclare the same type in the same file:

  type User = { id: string; name: string }; ❌ ❌ (This will throw a duplicate identifier error)

  ✅ IN REAL PROJECTS:

  You’d go back to 'user.ts' and modify the original 'User' type directly:

  // user.ts (updated version)

  export type User = {

    id: string;        // ✅ Refactored: changed from number to string

    name: string;

  };

  Because OptionalUser uses `keyof User` and `User[K]`, it auto-updates!

\*/

// We'll cover generics in detail in an upcoming chapter.

// ✅ Generic mapped type to convert any type's values to boolean

type Flags<T> = {

  // 📌 T is a generic placeholder — it can be any object type (User, Admin, Product, etc.)

  // 🔁 [K in keyof T] loops through each key of type T (like 'id', 'name', 'isActive')

  // 🧠 Instead of using the original type for each key, we set it to boolean

  [K in keyof T]: boolean;

};

// ⚙️ Now let's use that generic mapped type on the User type

type UserFlags = Flags<User>; // 🎯 This becomes:

// type UserFlags = {

//   id: boolean;

//   name: boolean;

//   isActive: boolean;

// }

const permissions: UserFlags = {

  id: true,

  name: false,

  isActive: true,

  // ✅ All fields are now booleans, thanks to the generic logic above

};

## Generic use cases of mapped types in our example

## ✅ 0. Base Type (Shared)

ts

CopyEdit

// 🔹 Original object type

type User = {

id: number;

name: string;

email: string;

isActive: boolean;

};

## ✅ 1. Make All Fields Optional

ts

CopyEdit

// 🔹 Mapped type: all fields become optional

type Partial<T> = {

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

};

/\*

Result:

type Partial<User> = {

id?: number;

name?: string;

email?: string;

isActive?: boolean;

}

\*/

const user: Partial<User> = {

name: "Likan" // ✅ valid — rest are optional

};

## ✅ 2. Make All Fields Readonly

ts

CopyEdit

// 🔹 Mapped type: all fields become readonly

type Readonly<T> = {

[K in keyof T]: Readonly<T[K]>;

};

/\*

Result:

type Readonly<User> = {

readonly id: number;

readonly name: string;

readonly email: string;

readonly isActive: boolean;

}

\*/

const user: Readonly<User> = {

id: 1,

name: "Likan",

email: "likan@example.com",

isActive: true

};

// user.id = 2; ❌ Error: Cannot assign to 'id' because it is readonly

## ✅ 3. Filter Only String Keys

ts

CopyEdit

// 🔹 Mapped type: keep only fields whose value is string

type StringOnly<T> = {

[K in keyof T as T[K] extends string ? K : never]: T[K];

};

/\*

Result:

type StringOnly<User> = {

name: string;

email: string;

}

\*/

const user: StringOnly<User> = {

name: "Likan", // ✅

email: "likan@example.com" // ✅

// id: 1 ❌ not allowed

// isActive: true ❌ not allowed

};

## ✅ 4. Prefix All Keys (e.g. user\_)

ts

CopyEdit

// 🔹 Mapped type: add prefix to all keys

type WithPrefix<T> = {

[K in keyof T as `user\_${string & K}`]: T[K];

};

/\*

Result:

type WithPrefix<User> = {

user\_id: number;

user\_name: string;

user\_email: string;

user\_isActive: boolean;

}

\*/

const user: WithPrefix<User> = {

user\_id: 1,

user\_name: "Likan",

user\_email: "likan@example.com",

user\_isActive: true

};

## ✅ 5. Nullable String Fields Only

ts

CopyEdit

// 🔹 Mapped type: only string fields become string | null

type NullableStrings<T> = {

[K in keyof T as T[K] extends string ? K : never]: string | null;

};

/\*

Result:

type NullableStrings<User> = {

name: string | null;

email: string | null;

}

\*/

const user: NullableStrings<User> = {

name: null, // ✅

email: "likan@example.com" // ✅

// id: 1 ❌ not allowed

// isActive: true ❌ not allowed

};

## ✅ 6. Remove Specific Key (id)

ts

CopyEdit

// 🔹 Mapped type: remove the 'id' key

type RemoveId<T> = {

[K in keyof T as K extends "id" ? never : K]: T[K];

};

/\*

Result:

type RemoveId<User> = {

name: string;

email: string;

isActive: boolean;

}

\*/

const user: RemoveId<User> = {

name: "Likan", // ✅

email: "likan@example.com", // ✅

isActive: true // ✅

// id: 1 ❌ Error: 'id' is not allowed

};

## ✅ 7. Add Boolean Flags from Keys (e.g. has\_id)

ts

CopyEdit

// 🔹 Mapped type: create flags for each key

type Flags<T> = {

[K in keyof T as `has\_${string & K}`]: boolean;

};

/\*

Result:

type Flags<User> = {

has\_id: boolean;

has\_name: boolean;

has\_email: boolean;

has\_isActive: boolean;

}

\*/

const user: Flags<User> = {

has\_id: true,

has\_name: false,

has\_email: true,

has\_isActive: true

};

## Full Power of Mapped Types

| **Feature** | **Benefit** |
| --- | --- |
| ✅ DRY | No manual repetition of properties |
| ✅ Safe Refactoring | Auto-updates with base type changes |
| ✅ Generic Reuse | Apply to any type (e.g. Flags<T>) |
| ✅ Transform Modifiers | Add/remove readonly, optional, required |
| ✅ Rename Keys | Use as to change key names |
| ✅ Conditional Types | Change value types conditionally |
| ✅ Filter Keys | Exclude/include keys based on value type |

**✅ 3. Basic Syntax**

type MyMappedType<T> = {

[Key in keyof T]: T[Key];

};

🔍 **Breakdown**:

* Key in keyof T → loop over each property in T
* T[Key] → get the value type of that property
* You can modify the keys, values, or even add modifiers (?, readonly)

// 🎯 Base type

type User = {

id: number;

name?: string; // optional

readonly isActive: boolean; // readonly

};

// ✅ 1. Remove optional (make all required)

type RequiredUser = {

[K in keyof User]-?: User[K]; // ❗ -? removes optional modifier

};

// Result:

// {

// id: number;

// name: string; ✅ now required

// readonly isActive: boolean;

// }

// ✅ 2. Remove readonly (make all mutable)

type MutableUser = {

-readonly [K in keyof User]: User[K]; // 🔓 -readonly removes readonly

};

// Result:

// {

// id: number;

// name?: string;

// isActive: boolean; ✅ now mutable

// }

// ✅ 3. Add both: readonly and optional

type ReadonlyOptionalUser = {

readonly [K in keyof User]?: User[K]; // 🔒❓ adds both readonly + optional

};

// Result:

// {

// readonly id?: number;

// readonly name?: string;

// readonly isActive?: boolean;

// }

### ✅ Mapped Type Modifiers

1. **?**  
   👉 Adds optional to all properties

ts

CopyEdit

type Optional<T> = { [K in keyof T]?: T[K] };

1. **-?**  
   👉 Removes optional and makes all properties required

ts

CopyEdit

type Required<T> = { [K in keyof T]-?: T[K] };

1. **readonly**  
   👉 Makes all properties readonly

ts

CopyEdit

type Readonly<T> = { readonly [K in keyof T]: T[K] };

1. **-readonly**  
   👉 Removes readonly and makes all properties mutable

ts

CopyEdit

type Mutable<T> = { -readonly [K in keyof T]: T[K] };

✅ Real-World Example: API Response Wrapper Using Mapped Types

### // 🔹 Generic mapped type to wrap each property with { data, loaded }

### type ApiResponse<T> = {

### [K in keyof T]: {

### data: T[K];

### loaded: boolean;

### };

### };

### // 🔹 Using it inline with API response shape

### const state: ApiResponse<{

### user: string;

### posts: string[];

### }> = {

### user: {

### data: "Likan", // ✅ string

### loaded: true

### },

### posts: {

### data: ["Post 1", "Post 2"], // ✅ string[]

### loaded: false

### }

### };

### /\*

### 🔍 Final resolved type of ApiResponse<...> becomes:

### type State = {

### user: {

### data: string;

### loaded: boolean;

### };

### posts: {

### data: string[];

### loaded: boolean;

### };

### }

### Note –

### type ApiResponse<T> = {

### [K in keyof T]: { // 🔁 Loop through each key in T ("user" | "posts")

### data: T[K]; // 🧠 Keep the original value type

### loaded: boolean; // ➕ Add a loading state

### };

### };

### /\*

### 🔍 Final Result of this type

### type Result = {

### user: {

### data: string;

### loaded: boolean;

### };

### posts: {

### data: string[];

### loaded: boolean;

### };

### }

### \*/

### \*/✅ Mapped Type Use Cases

* **Partial / Readonly / Required**  
  👉 Used to make all properties optional, readonly, or required in one go.
* **API Response Wrapping**  
  👉 Used to wrap each field with extra info like data and loaded.
* **Permissions / Feature Flags**  
  👉 Turns every property into a true/false toggle — great for access control.
* **Key Remapping with as**  
  👉 Used to rename keys (like add user\_ before every field).
* **Make Values Nullable**  
  👉 Adds null to all fields — helpful when data might be missing.
* **Filter Keys by Type**  
  👉 Keeps only the fields of a certain type (like only string fields).

**🧠 Interview Questions**

**Q1: What are Mapped Types in TypeScript?**  
Mapped types let you create new types by transforming the keys of an existing type using keyof and a loop-like syntax.

**Q2: What does [K in keyof T] mean?**  
It means: "loop through each key in T and do something with the value".

**✅ Difference Between Partial<T> and Custom Mapped Type Like Flags<T>**

ts

CopyEdit

// 🎯 Base type

type User = {

id: number;

name: string;

isActive: boolean;

};

**✅ 1. Built-in Partial<T>**

ts

CopyEdit

type OptionalUser = Partial<User>;

/\*

Internally becomes:

type OptionalUser = {

id?: number;

name?: string;

isActive?: boolean;

}

\*/

const user1: OptionalUser = {

name: "Likan" // ✅ allowed — all fields are optional

// id, isActive are not required

};

🔍 Partial<T>:

* Loops through keys in T
* Adds ? (optional modifier)
* Keeps the **original value types**

**✅ 2. Custom Mapped Type: Flags<T>**

ts

CopyEdit

type Flags<T> = {

[K in keyof T]: boolean; // ✅ transforms all value types into boolean

};

type UserFlags = Flags<User>;

/\*

Expands to:

type UserFlags = {

id: boolean;

name: boolean;

isActive: boolean;

}

\*/

const permissions: UserFlags = {

id: true,

name: false,

isActive: true // ✅ all fields are required and must be boolean

};

🔍 Flags<T>:

* Loops through keys in T
* Replaces all values with boolean
* Keeps keys required unless you explicitly add ?

**Q4: How do mapped types help with refactoring?**  
Mapped types help with refactoring by dynamically reflecting changes in the base type, so dependent types auto-update without manual edits.

// 🔹 Step 1: Base type

type User = {

id: number;

name: string;

isActive: boolean;

};

// 🔹 Step 2: Generic mapped type that transforms value types to boolean

type BooleanFlags<T> = {

[K in keyof T]: boolean;

};

// 🔹 Step 3: Apply mapped type dynamically

type UserFlags = BooleanFlags<User>;

/\*

At this point, UserFlags becomes:

type UserFlags = {

id: boolean;

name: boolean;

isActive: boolean;

}

\*/

// 🔹 Step 4: Refactor original User type (rename a key)

type User = {

id: number;

name: string;

status: boolean; // 🔁 renamed 'isActive' → 'status'

};

// 🔹 Step 5: UserFlags automatically reflects the updated shape

// No need to touch BooleanFlags or UserFlags

const flags: UserFlags = {

id: true,

name: false,

status: true // ✅ auto-updated — no 'isActive' anymore

// isActive: false ❌ Error: no such property

};

**Q5: Can I add modifiers in mapped types?**  
Yes! You can add readonly, ?, and even use conditional types to manipulate values.

Full Example: Adding Custom Properties to a Mapped Type Using Intersection (&)

// 🎯 Step 1: Base type

type User = {

id: number;

name: string;

isActive: boolean;

};

// ✅ Step 2: Create a mapped type and add extra fields using intersection (&)

type OptionalUserWithMeta = {

[K in keyof User]?: User[K]; // 🔁 Mapped type: make all User fields optional

} & {

updatedAt: string; // 🆕 Custom field 1

createdBy: string; // 🆕 Custom field 2

};

// 🔗 We're using intersection (&) to combine both the mapped type and extra fields

// ✅ Step 3: Example usage

const user: OptionalUserWithMeta = {

// Optional fields from User

name: "Likan",

// Required custom metadata fields

updatedAt: "2025-06-13",

createdBy: "admin"

};

Full Example: Using Union (|) with Mapped Type

// 🎯 Step 1: Base type

type User = {

id: number;

name: string;

isActive: boolean;

};

// ✅ Step 2: Mapped type (all optional)

type OptionalUser = {

[K in keyof User]?: User[K]; // 🔁 All properties optional

};

// ✅ Step 3: Add extra properties using UNION (|), not intersection

type OptionalUserOrMeta =

| OptionalUser // 🔁 one option: optional user fields

| { // OR this option: metadata only

updatedAt: string;

createdBy: string;

};

// ✅ Step 4: Usage Example

const data1: OptionalUserOrMeta = {

name: "Likan" // ✅ Valid — matches OptionalUser

};

const data2: OptionalUserOrMeta = {

updatedAt: "2025-06-13",

createdBy: "admin" // ✅ Valid — matches metadata object

};

const data3: OptionalUserOrMeta = {

name: "Likan",

updatedAt: "2025-06-13"

// ❌ Error in strict mode: This object includes props from both types

// Union doesn't allow combining unless both share those keys

};