# 📘 Chapter 14: Utility Types in TypeScript

## ✅ 1. Concept Overview

**Utility Types** are built-in generic types provided by TypeScript to simplify common type transformations.

They let you:

* Transform existing types into new types without repeating yourself (DRY)
* Add/remove optional, readonly, required modifiers
* Filter, extract, or combine types dynamically

They work especially well with advanced features like keyof, mapped types, and generics.

## ✅ 2. Key Benefits

* DRY code — reuse existing types
* Type-safe transformations
* Cleaner and more expressive type declarations
* Great for large-scale apps and API modeling

## ✅ 3. Commonly Used Utility Types

### 1. Partial<T>

➡️ Makes **all properties optional**

T must be a **single object type**.

Example -

### // 👤 Original type

### type Person = {

### name: string;

### age?: number;

### email?: string;

### };

### // 🔹 T = Person

### // Partial<T> makes \*\*all\*\* fields optional (even ones that were required)

### type PartialPerson = Partial<Person>;

### // Result:

### // {

### // name?: string;

### // age?: number;

### // email?: string;

### // }

### const person3: PartialPerson = {

### age: 25

### // ✅ All properties are now optional — we can pass any or none

### };

### 2. Required<T>

➡️ Makes **all properties required**

T must be a **single object type**.

Example -

// 👤 Original type

type Person = {

name: string;

age?: number; // optional

email?: string; // optional

};

// 🔹 T = Person

// Required<T> removes all optional modifiers (everything becomes required)

type RequiredPerson = Required<Person>;

// Result:

// {

// name: string;

// age: number;

// email: string;

// }

const person2: RequiredPerson = {

name: "Likan",

age: 30,

email: "likan@example.com" // ✅ must provide all fields now

};

const badPerson: RequiredPerson = {

name: "Aman"

};

// ❌ Error: Property 'age' and 'email' are missing — now they’re required

### 3. Readonly<T>

➡️ Makes **all properties immutable**

T must be a single object type

Example -

### // 👤 Original type

### type Person = {

### name: string;

### age?: number;

### email?: string;

### };

### // 🔹 T = Person (our full type)

### // Readonly<T> makes ALL properties immutable (can't be reassigned)

### type ReadonlyPerson = Readonly<Person>;

### // Result:

### // {

### // readonly name: string;

### // readonly age?: number;

### // readonly email?: string;

### // }

### const person1: ReadonlyPerson = {

### name: "Likan",

### age: 30,

### email: "likan@example.com"

### };

### person1.name = "Aman"; // ❌ Error: 'name' is read-only

### 4. Pick<T, K>

➡️ Creates a new type by **selecting specific properties** from a type

In Pick<T, K>,  
✅ T must be a **single object type**,  
✅ K can be **one or more keys** from that type.

Example -

### // 👤 Original type with all properties

### type Person = {

### name: string;

### age: number;

### email: string;

### };

### // 🔹 Pick<T, K> (multiple keys)

### // T = Person (the full original type)

### // K = "name" | "email" (the keys we want to keep from Person)

### type ContactInfo = Pick<Person, "name" | "email">;

### // Now ContactInfo = { name: string; email: string }

### const contact: ContactInfo = {

### name: "Likan", // ✅ allowed

### email: "likan@example.com" // ✅ allowed

### // age: 30 ❌ Error: age is not part of ContactInfo

### };

### Pick<T, K> (Single Key)

ts

CopyEdit

type Person = {

name: string;

age: number;

email: string;

};

// 🔹 Pick only the "name" property from Person

type NameOnly = Pick<Person, "name">;

const user: NameOnly = {

name: "Likan" // ✅ Only "name" is allowed here

};

### 5. Omit<T, K>

➡️ Creates a new type by **excluding specific properties** from a type

### // 👤 Original type with all properties

### type Person = {

### name: string;

### age: number;

### email: string;

### };

### // 🔹 Omit<T, K>

### // T = Person (original type)

### // K = "email" (the key we want to remove from Person)

### type WithoutEmail = Omit<Person, "email">;

### // Now WithoutEmail = { name: string; age: number }

### const partialPerson: WithoutEmail = {

### name: "Likan", // ✅ allowed

### age: 30 // ✅ allowed

### // email: "x@y.com" ❌ Error: email is not part of WithoutEmail

### };

### Example – Omit multiple keys

ts

CopyEdit

type Person = {

name: string;

age: number;

email: string;

address: string;

};

// 🧼 Remove "email" and "address"

type BasicInfo = Omit<Person, "email" | "address">;

/\*

BasicInfo = {

name: string;

age: number;

}

\*/

const user: BasicInfo = {

name: "Likan",

age: 30

};

### T **must be a single object type**, but K **can be multiple keys** (as a union like "a" | "b").

### 6. Record<K, T>

Record<K, T> is a TypeScript utility type that **constructs an object type** where the **keys are from K** and **all values are of type T**.

### ✅ 1️⃣ ****Single K, Single T****

ts

CopyEdit

type Result1 = Record<"admin", boolean>;

/\*

{

admin: boolean;

}

\*/

const obj1: Result1 = {

admin: true // ✅ allowed

};

// ❌ Invalid: key must be "admin", value must be boolean

const invalid1: Result1 = {

admin: "yes" // ❌ string not allowed

// user: false // ❌ extra key not allowed

};

type Roles = "admin" | "user";

const permissions: Record<Roles, boolean> = {

admin: true,

user: false,

};

### ✅ 2️⃣ ****Multiple K, Single T****

ts

CopyEdit

type Result2 = Record<"admin" | "user" | "guest", boolean>;

/\*

{

admin: boolean;

user: boolean;

guest: boolean;

}

\*/

const obj2: Result2 = {

admin: true,

user: false,

guest: true // ✅ all good

};

// ❌ Invalid: missing or wrong key/value

const invalid2: Result2 = {

admin: true,

user: "no", // ❌ value must be boolean

// guest missing // ❌ all keys required

};

### ✅ 3️⃣ ****Single K, Multiple T (Union Type)****

ts

CopyEdit

type Result3 = Record<"status", "active" | "inactive" | "pending">;

/\*

{

status: "active" | "inactive" | "pending";

}

\*/

const obj3: Result3 = {

status: "pending" // ✅ allowed value

};

// ❌ Invalid value

const invalid3: Result3 = {

status: "done" // ❌ not part of union

};

### ⚠️ 4️⃣ ****Multiple K, Multiple T (Loose Union Type)****

ts

CopyEdit

type Result4 = Record<"port" | "debug", number | boolean>;

/\*

{

port: number | boolean;

debug: number | boolean;

}

\*/

const obj4: Result4 = {

port: 3000, // ✅ valid number

debug: true // ✅ valid boolean

};

// ⚠️ Still valid, but unsafe:

const looseValid: Result4 = {

port: true, // ⚠️ valid but likely incorrect

debug: 1 // ⚠️ valid but confusing

};

## ✅ 7. Exclude<T, U>

**Exclude<T, U> removes from T all types that also exist in U,**  
and returns a **new type made of the remaining types from T**.

### 🧠 Conceptual Definition:

ts

CopyEdit

Exclude<T, U>

* T: Union of types (input set)
* U: Types you want to remove from T

### 🔧 Basic Example

ts

CopyEdit

type A = Exclude<"a" | "b" | "c", "a">;

// Result: "b" | "c"

🧠 Removes "a" from the union.

### 🔄 Transformed Type:

ts

CopyEdit

type A = "b" | "c";

## 📘 Common Use Cases with Code & Result

### ✅ Case 1: Remove a single value

ts

CopyEdit

type T1 = Exclude<"admin" | "user" | "guest", "guest">;

// Result: "admin" | "user"

### ✅ Case 2: Remove multiple values (union in U)

ts

CopyEdit

type T2 = Exclude<"a" | "b" | "c" | "d", "a" | "c">;

// Result: "b" | "d"

### ✅ Case 3: Remove null and undefined (common in cleanup)

ts

CopyEdit

type Clean = Exclude<string | null | undefined, null | undefined>;

// Result: string

### ✅ Case 4: Exclude from numeric union

ts

CopyEdit

type Num = Exclude<1 | 2 | 3 | 4, 2 | 3>;

// Result: 1 | 4

### ✅ Case 5: Exclude from object union

ts

CopyEdit

type Shapes =

| { kind: "circle"; radius: number }

| { kind: "square"; size: number };

type ExcludeSquare = Exclude<Shapes, { kind: "square"; size: number }>;

// Keeps only the circle object type

## 🧠 One-liner Summary:

Exclude<T, U> removes all members of T that are assignable to U.  
Useful for **filtering unions**, removing "null", "undefined", or specific values.

## ✅ Allowed & ❌ Invalid Example

ts

CopyEdit

type Role = "admin" | "user" | "guest";

type Filtered = Exclude<Role, "guest">; // "admin" | "user"

const role: Filtered = "admin"; // ✅

const invalid: Filtered = "guest"; // ❌ "guest" was excluded

### ⚠️ Note:

* Only works on **unions**.

### If U includes something not in T, it’s simply ignored. What if U contains values that aren’t in T?

✅ No problem — they’re **just ignored**.

ts

CopyEdit

type A = Exclude<"a" | "b", "c">;

// "c" is not in T, so it's ignored

// Result: "a" | "b"

ts

CopyEdit

type B = Exclude<1 | 2 | 3, 4 | 5>;

// Result: 1 | 2 | 3

✅ Still works — nothing is removed, since U doesn’t overlap with T.

8. Extract<T, U>

Extract<T, U> extracts the common types between T and U and creates a new type including only those shared types.

### ✅ Case 1: Overlapping string unions

ts

CopyEdit

type T1 = "a" | "b" | "c";

type U1 = "a" | "c" | "x";

type Result = Extract<T1, U1>;

// ✅ "a" | "c" — common parts

### ✅ Case 2: Mixed with extra values in U

ts

CopyEdit

type T2 = "dog" | "cat";

type U2 = "cat" | "lion";

type Result = Extract<T2, U2>;

// ✅ "cat" — only overlap

### ✅ Case 3: With string | number | boolean

ts

CopyEdit

type T3 = string | number | boolean;

type U3 = string | boolean;

type Result = Extract<T3, U3>;

// ✅ string | boolean

### ✅ Case 4: Numeric unions

ts

CopyEdit

type T4 = 1 | 2 | 3 | 4;

type U4 = 3 | 4 | 5;

type Result = Extract<T4, U4>;

// ✅ 3 | 4

### ✅ Case 5: Object type unions (structural matching)

ts

CopyEdit

type T5 =

| { type: "circle"; radius: number }

| { type: "square"; side: number };

type U5 = { type: "circle"; radius: number };

type Result = Extract<T5, U5>;

// ✅ Only the circle object type is extracted

### ✅ Case 6: No overlap at all

ts

CopyEdit

type T6 = "a" | "b";

type U6 = "x" | "y";

type Result = Extract<T6, U6>;

// ✅ Result: never

const value: A = "a"; // ❌ Error!

### 9. NonNullable<T>

➡️ Removes null and undefined from the type T

### ✅ Case 1: string | null | undefined

ts

CopyEdit

type A = NonNullable<string | null | undefined>;

// ✅ Result: string

### ✅ Case 2: number | null

ts

CopyEdit

type A = NonNullable<number | null>;

// ✅ Result: number

### ✅ Case 3: null | undefined

ts

CopyEdit

type A = NonNullable<null | undefined>;

// ✅ Result: never — everything is removed

### ✅ Case 4: Object + null

ts

CopyEdit

type User = { name: string } | null;

type A = NonNullable<User>;

// ✅ Result: { name: string }

### ✅ Case 5: Array or null

ts

CopyEdit

type A = NonNullable<string[] | null>;

// ✅ Result: string[]

### 10. ReturnType<T>

**➡️ Gets the return type of a function type T**

It’s like asking:  
🧠 *“What type does this function return?”*  
and creating a type from that.

### 🔧 Example:

ts

CopyEdit

function greet() {

return "Hello";

}

type GreetReturn = ReturnType<typeof greet>;

// ✅ GreetReturn = string

### ✅ Case 1: Function returning a string

ts

CopyEdit

function sayHi() {

return "Hi";

}

type A = ReturnType<typeof sayHi>;

// ✅ A = string

### ✅ Case 2: Function returning an object

ts

CopyEdit

function getUser() {

return { name: "Likan", age: 30 };

}

type UserType = ReturnType<typeof getUser>;

/\*

✅ UserType = {

name: string;

age: number;

}

\*/

### ✅ Case 3: Function returning union types

ts

CopyEdit

function getStatus(): "success" | "error" {

return "success";

}

type StatusType = ReturnType<typeof getStatus>;

// ✅ StatusType = "success" | "error"

### ✅ Case 4: Arrow functions

ts

CopyEdit

const add = (a: number, b: number) => a + b;

type AddReturn = ReturnType<typeof add>;

// ✅ AddReturn = number

### ✅ Case 5: Async function (Promise return)

async function fetchData() {

return [1, 2, 3];

}

type Data = ReturnType<typeof fetchData>;

// ✅ Data = Promise<number[]>

Want to extract the actual value from the Promise? Use Awaited<ReturnType<...>>.

type ResolvedData = Awaited<Data>; // ✅ ResolvedData = number[]

### 11. Parameters<T>

➡️ Extracts the parameter types of a function as a tuple.

It's like asking:

“What arguments does this function take?”  
and then creating a **tuple type** from them.

### 🔧 Example:

ts

CopyEdit

function add(a: number, b: string): void {}

type Params = Parameters<typeof add>;

// ✅ Params = [number, string]

**✅ Case 1: Regular function**

ts

CopyEdit

function greet(name: string, age: number) {}

type GreetParams = Parameters<typeof greet>;

// ✅ [string, number]

**✅ Case 2: Arrow function**

ts

CopyEdit

const sayHi = (name: string) => `Hi, ${name}`;

type SayHiParams = Parameters<typeof sayHi>;

// ✅ [string]

**✅ Case 3: Function with no params**

ts

CopyEdit

function getData() {}

type Params = Parameters<typeof getData>;

// ✅ []

**✅ Case 4: Using destructuring**

### // ✅ Case 1: Destructure in the function parameters (on the fly)

### function printInlineDestructured([name, age]: [string, number]) {

### // Already unpacked in the parameter

### console.log(`(Inline) ${name} is ${age} years old.`);

### }

### printInlineDestructured(["Likan", 30]);

### // ✅ Case 2: Destructure inside the function body

### function printManualDestructured(userData: [string, number]) {

### const [name, age] = userData; // Destructuring happens here manually

### console.log(`(Manual) ${name} is ${age} years old.`);

### }

### printManualDestructured(["Likan", 30]);

### ❌ Invalid Assignments (Examples that will throw errors)

Here’s how you'd get errors when calling either function:

ts

CopyEdit

// ❌ 1. Missing a value (only one item in the tuple)

printInlineDestructured(["Likan"]);

// Error: Source has 1 element(s) but target requires 2

printManualDestructured(["Likan"]);

// Same error

// ❌ 2. Wrong type in the tuple (age should be number)

printInlineDestructured(["Likan", "thirty"]);

// Error: Type 'string' is not assignable to type 'number'

printManualDestructured(["Likan", "thirty"]);

// Same error

// ❌ 3. Reversed order — wrong position

printInlineDestructured([30, "Likan"]);

// Error: Type 'number' is not assignable to type 'string'

printManualDestructured([30, "Likan"]);

// Same error

// ❌ 4. Extra values — not allowed in fixed-length tuple

printInlineDestructured(["Likan", 30, "extra"]);

// Error: Source has 3 element(s) but target only allows 2

printManualDestructured(["Likan", 30, "extra"]);

// Same error

### 12. InstanceType<T>

InstanceType<T> takes a class (or constructor function) and gives you the **type of the object** that would be created by new T().

**✅ Case 1: Normal class**

**// ✅ 1. Basic class**

**class User {**

**name: string = "Likan";**

**age: number = 30;**

**}**

**// ✅ 2. Get the instance type using InstanceType + typeof**

**type UserInstance = InstanceType<typeof User>;**

**// 🔍 UserInstance is now equal to: { name: string; age: number }**

**// ✅ 3. Create an instance using `new` — valid**

**const u1: UserInstance = new User();**

**// ✅ Works fine — new User() returns an object with shape: { name: string; age: number }**

**// ✅ 4. Use a plain object with matching structure — also valid**

**const u2: UserInstance = {**

**name: "Likan",**

**age: 30**

**};**

**// ✅ Allowed because the object structure matches the class instance**

**// ❌ 5. Object missing a required property — invalid**

**const u3: UserInstance = {**

**name: "Likan"**

**// ❌ Error: Property 'age' is missing**

**};**

**// ❌ 6. Object has wrong type — invalid**

**const u4: UserInstance = {**

**name: "Likan",**

**age: "thirty"**

**// ❌ Error: Type 'string' is not assignable to type 'number'**

**};**

**// ❌ 7. Object has extra unexpected property — invalid**

**const u5: UserInstance = {**

**name: "Likan",**

**age: 30,**

**role: "admin"**

**// ❌ Error: Object literal may only specify known properties**

**};**

**// ✅ 8. Class with private field**

**class Admin {**

**private id = 123;**

**name: string = "Likan";**

**role: string = "admin";**

**}**

**type AdminInstance = InstanceType<typeof Admin>;**

**// ✅ Valid: must use actual instance of Admin**

**const a1: AdminInstance = new Admin(); // ✅ works**

**// ❌ Not allowed: can't use object literal (missing private member)**

**const a2: AdminInstance = {**

**name: "Likan",**

**role: "admin"**

**// ❌ Error: Type is missing private member 'id'**

**};**

**✅ Case 2: Class with methods**

// ✅ 1. Define a class with a method

class Logger {

log(msg: string) {

console.log("LOG:", msg);

}

}

// ✅ 2. Extract the instance type from the class using typeof + InstanceType

type LoggerInstance = InstanceType<typeof Logger>;

// 🔍 LoggerInstance = { log(msg: string): void }

// ✅ 3. Valid: use an actual class instance

const logger1: LoggerInstance = new Logger();

logger1.log("Hello from logger1!"); // ✅ Works fine

// ✅ 4. Valid: use a manually created object with same structure

const logger2: LoggerInstance = {

log: (msg: string) => console.log("From literal:", msg)

};

logger2.log("Hello from logger2!"); // ✅ Works fine

// ❌ 5. Invalid: method name mismatch

const logger3: LoggerInstance = {

logMessage: (msg: string) => console.log(msg)

// ❌ Error: Property 'log' is missing

};

// ❌ 6. Invalid: wrong parameter type

const logger4: LoggerInstance = {

log: (msg: number) => console.log(msg)

// ❌ Error: Type '(msg: number) => void' is not assignable to '(msg: string) => void'

};

## 🔁 4. Before vs After - Example with Partial

❌ Without Utility Type

interface ManualPartialPerson {

name?: string;

age?: number;

}

✅ With Utility Type

type PartialPerson = Partial<Person>; // cleaner and DRY

## 🌍 5. Real-World Use Cases

* Building form models with optional inputs: Partial<FormFields>
* Fetching only necessary data: Pick<User, "id" | "name">
* Immutable Redux state slices: Readonly<State>
* API input/output validation types
* Modeling role-based access maps: Record<Role, boolean>

## ⚠️ 6. Limitations

* Overuse can make types less readable
* Inference may be confusing in deeply nested types
* Doesn’t replace clear modeling — still requires thoughtful design

## 💡 7. One-Liner Summaries

* Partial<T>: Makes all props optional
* Required<T>: Makes all props required
* Readonly<T>: Makes props immutable
* Pick<T, K>: Select only some props
* Omit<T, K>: Remove some props
* Record<K, T>: Key-value map with strict keys
* Exclude<T, U>: Remove matching types
* Extract<T, U>: Keep matching types
* NonNullable<T>: Remove null/undefined
* ReturnType<T>: Infer return of a function
* Parameters<T>: Infer params of a function
* InstanceType<T>: Get type of class instance

## 🎯 8. Interview Questions

### Q1. What does Partial<T> do?

**A:** It makes all properties of T optional.

### Q2. Difference between Pick and Omit?

**A:** Pick keeps selected keys, Omit removes specified keys.

### Q3. How would you type a role-permissions map?

**A:** Record<Role, boolean>

### Q4. How do you get function parameters as a tuple?

**A:** Use Parameters<typeof fn>

### Q5. How to exclude null and undefined from a union?

**A:** Use NonNullable<T>