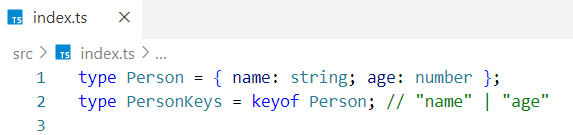
Chapter 12: keyof, typeof, & Indexed Access Types

***1. What is keyof ?***

The keyof operator **takes an object type** and **returns a union of its keys** as string literals.



***2. Key Benefits of keyof***

## ✅ 1. DRY (Don’t Repeat Yourself)

### ❌ Bad Approach (Not DRY) – without Key of

type User = {

id: number;

name: string; // 📍 Location #1 — key "name" is defined in the type

email: string;

};

// ❌ Hardcoded "name" — violates DRY

function getUserName(obj: User): string {

return obj["name"];

// 📍 Location #2 — key "name" is hardcoded again here

// ❌ DRY Violation: you're repeating the same key in two places manually

// ➤ If "name" is renamed in the type, you must update it here too — risky & not scalable

}

const user: User = {

id: 1,

name: "Likan",

email: "likan@example.com"

};

console.log(getUserName(user));

**Code Breaks Silently - without Key of –**

*// ✅ Type definition*

*type User = {*

*id: number;*

*name: string;*

*email: string;*

*};*

*// ❌ This is a separate object — it is NOT typed as `User` and has a different structure*

*const userDetailsRaw = {*

*id: 1,*

*fullName: "Likan Mishra", // ❌ Doesn't match expected `User` shape*

*email: "likan@example.com"*

*};*

*// 🧠 TypeScript infers this as: { id: number; fullName: string; email: string }*

*// It is NOT linked to our `User` type in any way*

*// ✅ Function that expects a strictly typed User object*

*function getUserName(user: User): string {*

*return user["name"];*

*// ✅ No TypeScript error here*

*// ✅ Because parameter `user` is explicitly declared as type `User`*

*// ✅ And in the `User` type, we do have a key `name` of type `string`*

*// ✅ So TS assumes this key is always present and valid*

*// ❗ Important Point:*

*// You might expect TS to validate the object shape \*inside\* the function,*

*// but that’s NOT where the check happens.*

*// TS only checks the object shape at the \*\*function call\*\* — not inside the function.*

*// This is a crucial TypeScript behavior to understand.*

*}*

*getUserName(userDetailsRaw);*

*// ✅ Compile-time ERROR:*

*// Argument of type '{ id: number; fullName: string; email: string; }'*

*// is not assignable to parameter of type 'User'.*

*// Property 'name' is missing in type...*

*// ❌ If you had written:*

*// getUserName(userDetailsRaw as User);*

*// ❌ Then TS would trust it blindly, no error,*

*// ❌ But at runtime → `user["name"]` would be undefined (silent bug) . code breaks silently here*

How keyof Comes to the Rescue — Prevents DRY Violation & Silent Failures ?

***// ✅ Type definition***

***type User = {***

***id: number;***

***name: string;***

***email: string;***

***};***

***// ✅ Reusable generic function using keyof***

***function getField<T, K extends keyof T>(obj: T, key: K): T[K] {***

***return obj[key];***

***// ✅ No hardcoded string***

***// ✅ Key must be one of the keys in the object type***

***}***

***// ✅ Valid object matching the User type***

***const user: User = {***

***id: 1,***

***name: "Likan",***

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

***};***

***console.log(getField(user, "name")); // ✅ Type-safe, DRY***

***console.log(getField(user, "email")); // ✅ Also safe***

***// ❌ Invalid key example***

***// console.log(getField(user, "fullName"));***

***// ❌ TS Error: Argument of type '"fullName"' is not assignable to parameter of type '"id" | "name" | "email"'***

✅ 2. Safe Refactoring

***// ✅ Generic function using keyof — safe & DRY***

***function getUserValue<T, K extends keyof T>(obj: T, key: K): T[K] {***

***return obj[key];***

***// ✅ Key is strongly tied to the object type***

***// ✅ Only keys from T are allowed — TS checks this at compile time***

***}***

***// 🔄 Suppose we update the User type***

***type User = {***

***id: number;***

***fullName: string; // 🔁 renamed from "name"***

***email: string;***

***};***

***const user: User = {***

***id: 1,***

***fullName: "Likan",***

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

***};***

***// ❌ This will now give a compile-time error because "name" no longer exists***

***const result = getUserValue(user, "name");***

***// ⛔ Error: Argument of type '"name"' is not assignable to parameter of type '"id" | "fullName" | "email"'.***

✅ 3. Intellisense (Autocomplete)

## ✅ 3. Intellisense (Autocomplete)

ts

CopyEdit

type User = {

id: number;

name: string;

email: string;

};

const user: User = {

id: 1,

name: "Likan",

email: "likan@example.com"

};

// Hover or start typing below:

const value = getUserValue(user, "");

// 🧠 IDE will suggest only valid keys: "id", "name", "email"

// ❌ If you type "username", TS will throw an error

// Example

const name = getUserValue(user, "name"); // ✅ works fine

✅ Thanks to K extends keyof T, TypeScript gives **valid key suggestions** in your editor.

## ✅ 4. Generic Utility

ts

CopyEdit

// 🔹 Define another type

type Product = {

title: string;

price: number;

inStock: boolean;

};

const product: Product = {

title: "Laptop",

price: 1000,

inStock: true

};

// 🔁 Reuse the same generic function

const title = getUserValue(product, "title"); // ✅ "Laptop"

const price = getUserValue(product, "price"); // ✅ 1000

const stock = getUserValue(product, "inStock"); // ✅ true

✅ This proves the function is **generic** and works with any object shape — not just User.

***Main benefits of key of –***

| Feature | Verified By Code Section |
| --- | --- |
| DRY | Compared hardcoded vs keyof-based function |
| Safe Refactoring | Renamed key and showed type-safe error |
| Intellisense | IDE shows valid suggestions only |
| Generic Utility | Worked for both User and Product types |
|  |  |

**Before keyof -** Unsafe Version — getValue()- TypeScript accepts just any key.

// ❌ UNSAFE FUNCTION

function getValue(obj: any, key: string) {

return obj[key]; // ⚠️ No type check, allows any string

}

// 🧪 TEST CASES for UNSAFE

type User = {

id: number;

name: string;

email: string;

};

const user: User = {

id: 1,

name: "Likan",

email: "likan@example.com",

};

// ✅ Valid key — works fine

const result1 = getValue(user, "name"); // OK, returns "Likan"

// ❌ Invalid key — no compile error, but undefined

const result2 = getValue(user, "phone"); // ❌ No type error, but runtime bug

console.log("❌ Unsafe - valid key:", result1); // "Likan"

console.log("❌ Unsafe - invalid key:", result2); // undefined (bad!)

**After keyof -** Safe Version — getValueSafe() - Uses keyof to ensure the key exists on the object.

**// ✅ SAFE FUNCTION**

**function getValueSafe<T, K extends keyof T>(obj: T, key: K): T[K] {**

**return obj[key]; // ✅ Key must be valid on obj**

**}**

**// 🧪 TEST CASES for SAFE VERSION**

**type User = {**

**id: number;**

**name: string;**

**email: string;**

**};**

**const user: User = {**

**id: 1,**

**name: "Likan",**

**email: "likan@example.com",**

**};**

**// ✅ Valid key - TypeScript ensures correctness**

**const safeResult1 = getValueSafe(user, "id"); // ✅ number**

**const safeResult2 = getValueSafe(user, "email"); // ✅ string**

**console.log("✅ Safe - ID:", safeResult1); // 1**

**console.log("✅ Safe - Email:", safeResult2); // "likan@example.com"**

**// ❌ Invalid key - COMPILE TIME ERROR**

**// const safeResult3 = getValueSafe(user, "phone");**

**// ❌ Error: Argument of type '"phone"' is not assignable to parameter of type '"id" | "name" | "email"'**

In the safe version:

ts

CopyEdit

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

return obj[key];

}

Here, K extends keyof T means:

* The key you pass **must** be one of the actual property names (keys) of the object type T.

So if T = User, then K can **only be** "id" | "name" | "email".

So this is fine:

ts

CopyEdit

getValueSafe(user, "name"); // ✅

But this causes a compile-time error:

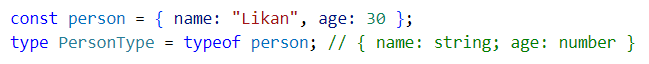
ts

CopyEdit

getValueSafe(user, "phone"); // ❌

***3. What is typeof?***

The typeof operator lets you get the type of a variable or function.



***4.Why use typeof?***

// 🔹 STEP 1: Define the original config object

const config = {

appName: "MyApp",

port: 8080,

debug: true

};

// ✅ 1. Reuse full shape using typeof

type ConfigType = typeof config;

// Same as:

// type ConfigType = {

// appName: string;

// port: number;

// debug: boolean;

// }

// ✅ 2. Use the inferred type for another variable

const devConfig: ConfigType = {

appName: "MyApp",

port: 3000,

debug: false

};

// ❌ 3. Try assigning wrong type to prove type safety

const brokenConfig: ConfigType = {

appName: "Oops", // ✅ correct type: string

port: "wrong", // ❌ ERROR: string is NOT assignable to number

// TypeScript compares this value with the original config:

// port: number ← from typeof config

// So if we pass a string, it throws a compile-time error

debug: true // ✅ correct type: boolean

};

// ✅ 4. Reuse it in function params

function startServer(cfg: ConfigType) {

console.log(`Starting ${cfg.appName} on port ${cfg.port}`);

}

startServer(devConfig); // ✅ type-safe

### Key Benefits of typeof in TypeScript

1. **Reuse Type** 📦  
   Get the full type shape of a variable (typeof config) without redefining it.
2. **DRY Code** 🧼  
   Avoid repeating type structures — change once, update everywhere.
3. **Type Safety** 🛡️  
   Ensures all variables or functions using that type follow the correct shape.

### ❌ WITHOUT typeof — Manually Repeating Types

ts

CopyEdit

// Define a user object

const user = {

name: "Likan",

age: 30,

};

// ❌ Function WITHOUT typeof

// You manually write out the object structure here

function greetManual(person: { name: string; age: number }) {

// Access the properties of the manually-typed object

return `Hi ${person.name}, you're ${person.age} years old.`;

}

// ✅ Works fine

console.log(greetManual({ name: "Likan", age: 30 }));

// ⚠️ Problem: If you add/remove/change properties in `user`, you have to update the type here too

### ✅ WITH typeof — Reuse Type from Variable

ts

CopyEdit

// Define a user object

const user = {

name: "Likan",

age: 30,

};

// ✅ Function WITH typeof

// Instead of repeating the shape, we infer it from the `user` object

function greetAuto(person: typeof user) {

// TS knows person.name is string, person.age is number

return `Hello ${person.name}, you are ${person.age} years old.`;

}

// ✅ Safe and clean — uses the actual type of `user`

console.log(greetAuto(user));

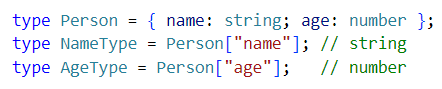
// 🧠 Benefits:

// - DRY: no repeated type

// - Safe: auto-updates if `user` structure changes. It means that **if you change the structure of the user object**, you **don’t need to manually update the type elsewhere**

***5. Indexed Access Types (T[K])***

Use T[K] to extract the type of a property from an object.



Real-World Use Case of keyof, typeof, and T[K] in One Code Snippet

### 🏷️ Scenario: Dynamic Form Field Reader

A type-safe utility function to **dynamically access fields** from an object like form data.

// 🔹 Step 1: Define a real-world object

const formData = {

name: "Likan",

age: 30,

email: "likan@example.com"

};

// 🔹 Step 2: Generic function to get any field's value

function getFormField<

T, // T is the type of the entire object (e.g. formData)

K extends keyof T // K must be a key from T (i.e. 'name' | 'age' | 'email')

>(data: T, fieldName: K): T[K] {

return data[fieldName]; // ✅ Indexed Access Type returns the correct type

}

// 🔹 Step 3: Use typeof to avoid repeating the shape

type FormType = typeof formData;

// FormType = { name: string; age: number; email: string }

// 🔹 Step 4: Test with real data

const name = getFormField<FormType, "name">(formData, "name"); // inferred as string

const age = getFormField(formData, "age"); // inferred as number

const email = getFormField(formData, "email"); // inferred as string

console.log(name.toUpperCase()); // Output: "LIKAN"

console.log(age + 1); // Output: 31

✅ **Case 1: <T, K> is Optional (Inferred Automatically)**

**// ✅ Case 1: Everything is strongly typed — TS can infer**

**type FormType = {**

**name: string;**

**age: number;**

**};**

**const formData: FormType = {**

**name: "Likan",**

**age: 30**

**};**

**// Generic function using T and K**

**function getField<T, K extends keyof T>(data: T, key: K): T[K] {**

**return data[key]; // 🔄 Returns type-safe value**

**}**

**// ✅ Inference works because:**

**// - formData has type FormType**

**// - "name" is a literal key of FormType**

**const name = getField(formData, "name"); // 🔁 T = FormType, K = "name" (inferred)**

**console.log(name.toUpperCase()); // Output: "LIKAN"**

### 🧠 Explanation:

* formData is explicitly typed → TypeScript knows the object structure
* "name" is a literal → TypeScript knows it's a key of FormType
* ✅ So, no need to pass <FormType, "name">

Case 2 — Required: TS Can’t Infer, You Must Pass <T, K>

**// ❌ Case 2: TypeScript can't infer from `any`, so manual typing is required**

**type FormType = {**

**name: string;**

**age: number;**

**};**

**const raw: any = {**

**name: "Likan",**

**age: 30**

**};**

**// Same generic function**

**function getField<T, K extends keyof T>(data: T, key: K): T[K] {**

**return data[key];**

**}**

**// ❌ In this case:**

**// - raw is of type `any` → no type info**

**// - TS can't safely infer T or K**

**// ✅ So you MUST specify them manually**

**const name = getField<FormType, "name">(raw, "name");**

**console.log(name.toUpperCase()); // Output: "LIKAN"**

### 🧠 Explanation:

* raw is any → TypeScript has zero info about its keys or values
* Without manually passing <FormType, "name">, TypeScript throws an error or infers everything as any
* ✅ Passing <T, K> explicitly fixes it

Interview Insights -

***Q1: What is keyof?***

It creates a union of all keys of a given object type. Example: keyof Person → "name" | "age"

### ❓ Q2: What’s the Difference Between typeof in JavaScript vs TypeScript?

#### 🔸 In **JavaScript**, typeof is a **runtime operator**.

That means it works while your code is **running** in the browser or Node.js. It returns a **string** representing the type of the value.

✅ Example:

js

CopyEdit

const age = 30;

console.log(typeof age); // "number"

So in JavaScript, typeof is used for:

* Checking types during execution
* Debugging logic
* Writing conditions (e.g., if (typeof x === "string"))

#### 🔹 In **TypeScript**, typeof is a **compile-time type operator**.

It doesn't return a string like JavaScript. Instead, it tells the TypeScript compiler to **extract the type** of a variable, function, or value — to use in type declarations.

✅ Example:

ts

CopyEdit

const user = {

name: "Likan",

age: 30

};

type UserType = typeof user;

// Now UserType is: { name: string; age: number }

So in TypeScript, typeof is used to:

* Avoid repeating object structures
* Make utility types reusable
* Get static types based on existing variables

**🧠 Summary in Plain Words:**

* In **JS**, typeof gives you the **type as a string** at runtime
* In **TS**, typeof gives you the **type as a type** at compile-time

***Q3: What is an indexed access type?***

T[K] lets you extract the type of a property from a type.  
Example: Person["name"] → string

***Q4: Can you use all 3 together?***

Yes! typeof to extract type from value, keyof to get valid keys, T[K] to access property types.