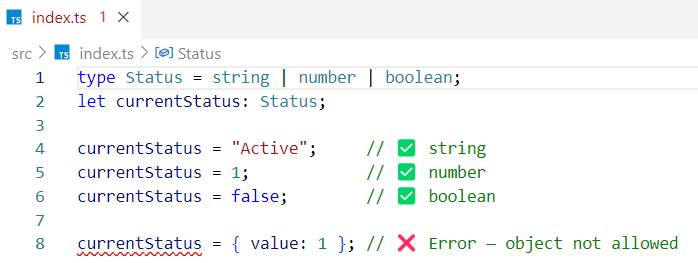
📘 CHAPTER 6: Union Types, Intersection Types & Literal Types

***1. What is a Union Type?***

A union type lets a variable hold more than one possible type.

Syntax: type <typeName> = <Type1> | <Type2> | <Type3>;

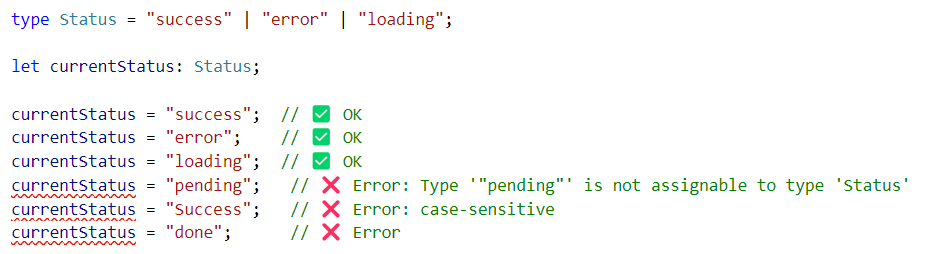
Example 1: type Status = string | number | boolean;



Example 2 -

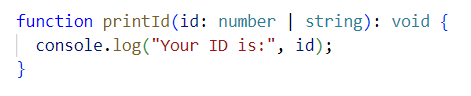
type Status = "success" | "error" | "loading";

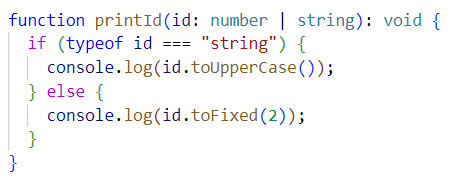
In this case, status can only be one of these three string literals. If a user tries to assign any value other than "success", "error", or "loading", TypeScript will throw a type error. It means: status can only be one of those three string values.



**💡 Union = OR**

***Union use Case - Function Accepting Multiple Types***





TypeScript forces you to check the type before using methods that only work on certain types.

In our example, id can be a number or a string (a union type).  
Since TypeScript doesn't know the exact type at runtime, you have to check the type first.

That’s why we write if (typeof id === "string") { ... } else { ... }

This is called type narrowing - it helps TypeScript understand the exact type inside each block so it can allow only the right methods.

For example -

* toUpperCase() only works on strings
* toFixed() only works on numbers

So, we narrow the type before using them ✅

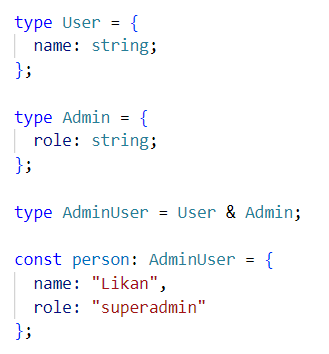
***2. What is an Intersection Type?***

An intersection type lets you **combine multiple types into one.**  
The resulting type must have **all the properties** from all the combined types.

Syntax: type <NewType> = Type1 & Type2 & Type3;

It means the variable must satisfy **Type1 and Type2 and Type3** - all together.

***Example 1: Combine Two Object Types***



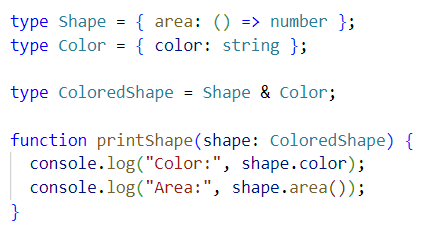
In an intersection type, all required properties from all types must be included.  
In our example, name is required from User, and role is required from Admin, so the person object must have **both** name and role.

***Example 2: Combine with Optional Properties***



In an intersection type, optional properties remain optional, but required properties must be present. In our example, email is optional from Contact, but username is required from Profile, so the acc object must have username, and email is optional.

***Example 3: Function Parameter with Combined Types***



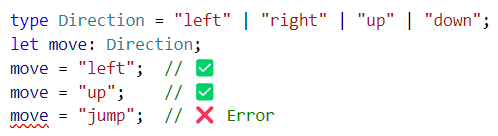
In this intersection type, ColoredShape combines both Shape and Color, so the shape object passed to printShape must have **both** a color property and an area() method - TypeScript enforces the presence of both.

**💡 Intersection = AND**

***3. What is a Literal Type***

Literal types restrict a variable to a specific value.

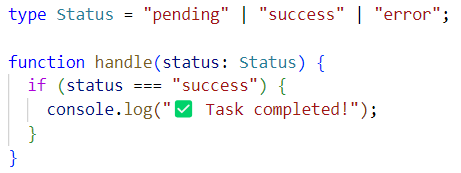
type Direction = "left" | "right" | "up" | "down";



Literal types are powerful when combined with unions.

How Literal Unions Mimic Enums in TypeScript ?

Union + literal types let you create enum-like behavior with better safety and less code.

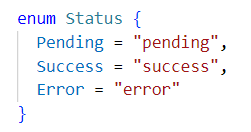


### *Why is this considered "enum-like" ?*

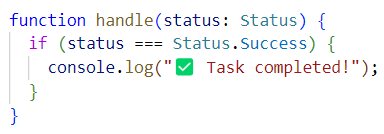
* It **restricts** the value to a **fixed set** of strings
* It provides **type safety** at compile time
* It supports **narrowing** in conditions (if (status === "success"))
* It’s shorter, cleaner, and often used over enums in modern TypeScript codebases

### *Enum (classic way):*

### An enum is a special type that lets you define a set of **named constants** - each name maps to a fixed value (like a string or number) so you can write safer and more readable code



Then you’d use it like



***PART 1 — Union of Two Types***

Goal: Create two different types and allow a variable/function to accept either one.

Step 1: Define two types -



🧠 Here -

* Admin has privileges
* User has email

Step 2: Create a union type



Step 3 : Applying a Union Type to an Object in TypeScript

If an object is of union type Person = Admin | User, it must **fully match** either the Admin structure or the User structure - not a mix.

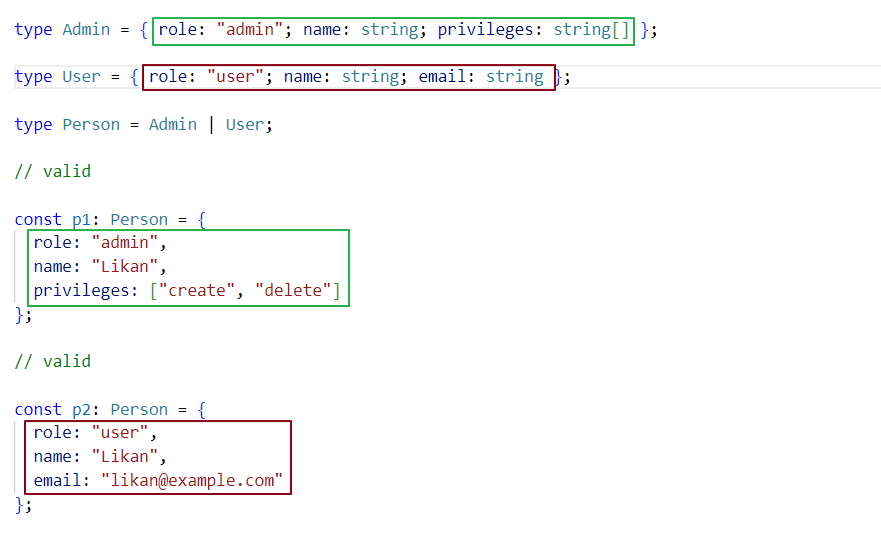
Only **common properties** like name and role are safely accessible without checking.

If we try to **access or include** an uncommon property (like email or privileges) **from the wrong type,** TypeScript will throw an error.

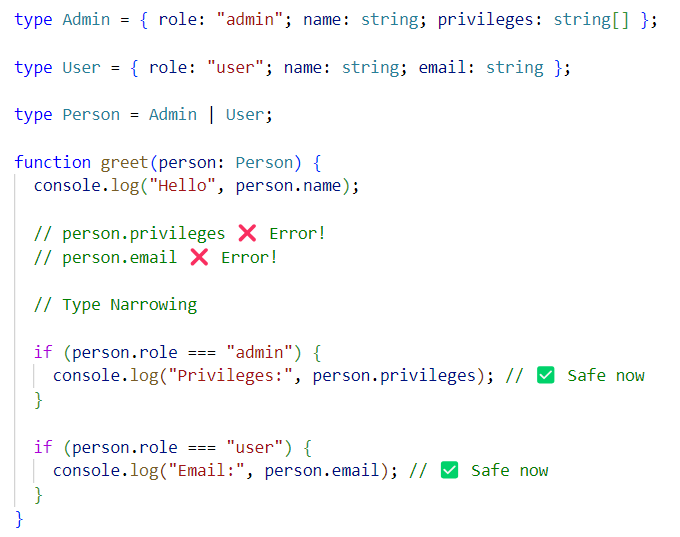
**Invalid -**



**Valid -**



Step 4: use Union Type with Type Narrowing in a Function



***Q: Why do we get an error when accessing person.privileges or person****.****email directly inside the function, but not inside the if block?***

Because inside the function, TypeScript only knows that person is of type Admin | User - a **union type**. It doesn’t know which specific type (Admin or User) it is at that moment. So, it only allows access to **common properties** shared by both types (like name or role).But privileges and email are **uncommon properties** - they exist only on one side of the union.

Trying to access them directly is risky because -

What if person is a User, and you're trying to access privileges (which only Admin has)?

That’s why TypeScript throws a **compile-time error** - to protect you from unsafe access.

To safely access uncommon properties, you must tell TypeScript which exact type you're working with like -



This process is called **type narrowing,** where we reduce a union type down to a more specific type using conditions.

***PART 2 — Intersection of Two Types***

Goal -Create two different types and ensure a variable/function accepts a value that satisfies both types at the same time.

Step 1: Define two types -



🧠 Here -

* Admin has privileges
* Employee has startDate

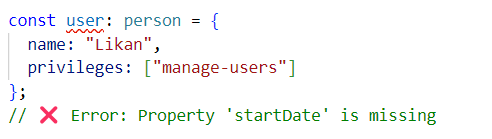
Step 2: Create an intersection type



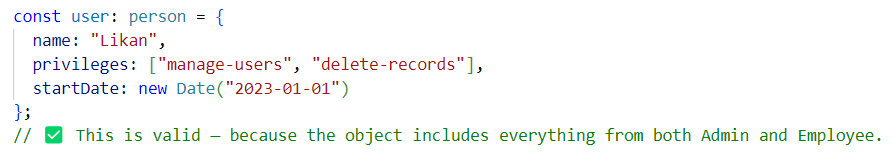
Step 3 : Applying an intersection Type to an Object in TypeScript

If an object is of **intersection type** Person = Admin & Employee, it must satisfy **both** the Admin and Employee structures at the same time. That means the object must include **all properties** from both types - including uncommon ones like privileges (from Admin) and startDate (from Employee).  
Unlike union types, **all properties from both sides are safely accessible** without type narrowing, because the object is guaranteed to have the full structure of both types.

**Invalid –**



**Valid –**

.

Interview insights -

***Q: Why use union instead of any?***

Union still gives safety & autocomplete but any disables type safety

***Q: What’s the practical use of intersection types***?

When you want a type to satisfy multiple roles. example - a user that is both an Admin & Employee

***Q: Can literal types help prevent bugs?***

Yes, they restrict values and give you autocomplete and compiler checks.

***Q: Output based question*** *.* ***What will this log?***



✅ Output: "Theme is: dark"  
❌ Error: Argument of type "blue" is not assignable to parameter of type 'Mode'

***Q: Why are literal types better than using string?***

They prevent typos, give better IntelliSense, and restrict what values are allowed — so your functions only accept valid input.

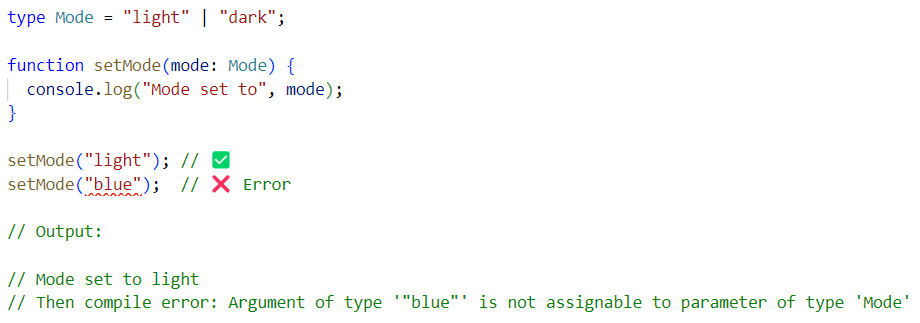
***Q: Output based question*** *.* ***What will this log? (Union)***



***Q: Output based question*** *.* ***What will this log? (Intersection)***



***Q: Output based question*** *.* ***What will this log? (Literal)***



Summary –

1. Union Type (A | B)  
   ✅ "Could be either A's properties or B's properties — not both."  
   ✅ You can only access properties common to both unless you use type narrowing.
2. Intersection Type (A & B)  
   ✅ "Must have all properties from both A and B."  
   ✅ It's a combination — the resulting object must fully satisfy both types.
3. Literal Types (e.g., "pending" | "success")  
   ✅ These are fixed, exact values - not structures like A or B.  
   🚫 A and B don’t apply here as 'types', because literal types don’t represent objects — they represent specific constant values.  
   ✅ You can think of them as distinct options, not separate type structures.