Union with Types and Tagged Union

In this lesson, we will see the union type and the tagged union.

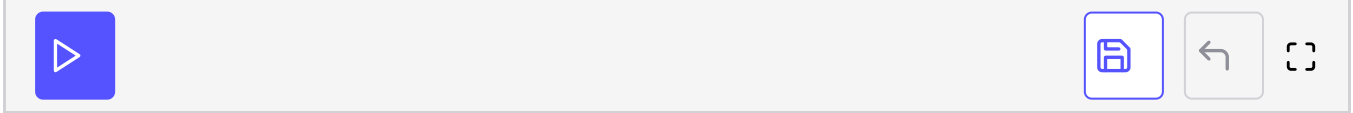
WE'LL COVER THE FOLLOWING

Union type

The union type is more common because it's often used to indicate that a variable can be one type or another. For example, a member could be a string or undefined. Union types are the strongly typed way to allow multiple types for a function, too.

If you have a function that can take a string or an object and depend on the type acting differently, a union can do the job. `any` type would also work, but the problem is that `any` allows everything while in reality, you want to have only a limited type of parameters. With `any` it wouldn't be until runtime that you could catch if something was not assigned correctly, but with the union, it's at compilation time. The return type of a function can also be a union. A function may return a specific object or undefined, for example, or return a number or a string.

```
let u1: string | boolean = true;
type UStringBoolean = string | boolean;
let u2: UStringBoolean = true;
```



The tagged union

Since TypeScript 2.0, the concept of the tagged union type has gained popularity. Imagine that you created a new type with the union of several interfaces. At some point in the execution of your system, there might be a need to narrow the type. This need may arise simply by wanting to perform a specific action that requires explicitly knowing the type. The possibility of being one of many types complicates the use of a union type. Using a particular feature of a specific type gives TypeScript a hint. Otherwise, only shared features are available.

TypeScript has many ways to determine which type within the union is the correct value. You can always check for specific members that are unique to each interface and use a method to tell TypeScript which interface it is. We will see that technique in the “type checking” section of this course. However, this requires creating a single method per interface. To avoid convoluted code, the use of tagged union type becomes handy.

```
interface InterfaceA {
    discriminant: "InterfaceA"; // This is not a string type, but Interfac
eA type
    m0: number;
}
```

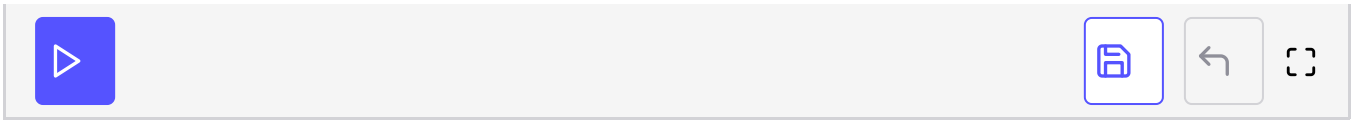
The concept is simple. Every interface needs a common field with the same name, but a different type. In the previous example, the discriminant is called `discriminant`, but it could be anything. Often, it is called `kind`. The type can be anything unique. A common use is to specify a type of a unique string to this discriminant field. It is good to note that the field type is not a string (which would allow all kinds of strings) but a specific one which indicates that only this string will be accepted.

The discriminant is built with something called a string literal type, as we will describe in this course. Up until now, nothing has been really specific to TypeScript. However, where TypeScript shines is when you compare the discriminant field to the type specified in one of the interfaces. TypeScript will automatically narrow down the type to the proper related interface.

For example, you can create a switch statement directly on the attribute using the discriminant field used for this purpose which is shared across all union types. The following example, at **line 14-15** is acting on the determination of

the type. Once in a case, the members of each specific interface are available without any casting.

```
interface InterfaceA {
    discriminant: "InterfaceA"; // This is not a string type, but InterfaceA type
    m0: number;
}
interface InterfaceB {
    discriminant: "InterfaceB"; // This is not a string type, but InterfaceB type
    m1: string;
}
interface InterfaceC {
    discriminant: "InterfaceC"; // This is not a string type, but InterfaceC type
    m2: string;
}
function unionWithDiscriminant(p: InterfaceA | InterfaceB | InterfaceC) {
    switch (
        p.discriminant // Only common members available
    ) {
        case "InterfaceA":
            console.log(p.m0); // Only InterfaceA members available
            break;
        case "InterfaceB":
            console.log(p.m1); // Only InterfaceB members available
            break;
        case "InterfaceC":
            console.log(p.m2); // Only InterfaceC members available
            break;
    }
}
```



The **line 18** variable `p` can access `m0` because the `case` is narrowing down the type to the `InterfaceA`.

TypeScript version 2.7 brings a change with classes and unions. From 2.7, union uses the same structure for classes in a union. Before 2.7, the type was narrowed.