Definition and Usages

In this lesson, we will set the groundwork for the upcoming lessons on the mapped type.

We'll cover the following

What is the mapped type?#

The mapped type allows for creating a new type from an existing one. Using the term *map* refers to pointing existing members to a new type by a custom logic that is unique to the mapper implementation.

A good example is to have an existing interface keep all the same members, but make them *optional* or *read-only*.

Advantages of the mapped type#

First advantage#

The mapped type has two advantages over the previous versions of TypeScript that didn't support it.

Before the mapped type, we needed to create an additional interface to

reflect the desired final state. This extra step pollutes the codebase of several

interfaces, depending on how you transform the interface and how many duplicates for each different interface need to be mapped.

Furthermore, for each original interface, you need to synchronize every other one each time the interface changes. It rapidly becomes hard to maintain.

The following code demonstrates how we needed two interfaces to have one read-only and another without a mapped type. This was prior to the introduction of a mapped type in the language. The code is trying to have two objects with the same attributes, one that can alter its values once initialized and another that should block any future assignments.

```
interface OriginalInterface {
    x: number;
    y: string;
}

interface ReadOnlyOriginalInterface {
    readonly x: number;
    readonly y: string;
}

let variable1: OriginalInterface = { x: 1, y: "2" };
let variable2: ReadOnlyOriginalInterface = { x: 1, y: "2" };
variable1.x = 2; // Can alter the value
// variable2.x = 2; // Cannot alter the value
```

Second advantage#

The second advantage is that without a mapped type, every interface requires a separate function to perform the transformation. Again, this gets out of control very fast. With the mapped type, you only need your transformation function and to define your mapped type. With the help of

generic, the transformation function is self-sufficient, taking [illegible]e interface and returning the mapped type definition which in turn, is also leveraging generic.

We will go into more detail about generic in future lessons. For the moment, let's simplify the concept of generic by having a reusable way to change the type of interface. The following code illustrates a simple `function` that converts one interface into another type.

```
// The interface that define a structure that allows re-assignment of values
interface OriginalInterface {
    x: number;
    y: string;
}

// The interface that defines a structure that allows assignment only at creating
interface ReadOnlyOriginalInterface {
    readonly x: number;
    readonly y: string;
}

let variable1: OriginalInterface = { x: 1, y: "2" };
let variable2: ReadOnlyOriginalInterface = { x: 1, y: "2" };

// A function that transform one object to the other type
function mapOriginalInterfaceToBeReadOnly(o: OriginalInterface): ReadOnlyOriginalInterface
    return o;
}

let variable3 = mapOriginalInterfaceToBeReadOnly(variable1);
// variable3.x = 2;
```

In the following lesson, we will see a more pragmatic scenario where mapped type can be employed to reduce the burden of having a function for every type we may convert. The code in this lesson did not use the mapped type and to convert an object to another interface required us to duplicate the interface and add the `readonly` to each field creating a function.

Let’s see a better way in the next lesson.

Back

Merging and Adding Functionality to E...

Next

Immutable Data with Readonly

Mark as Completed

Report an Issue