The Primitive Type never

In this lesson, you will see the type never, which is used to indicate that something must never happen.

The type `never` means that nothing occurs. It is used when a type guard cannot occur or in a situation where an exception is always thrown. There is a difference between `void` and `never`. A function that has the explicit return type of `never` won't allow your code to return `undefined`, which is different from a `void` function that allows code to return `undefined`.

```
1  function functionThrow(): never {
2      throw new Error("This function return never");
3  }
4
```

The `never` type is a subtype for every type. Hence, you can return `never` (for example, throwing an exception) when a return type is specified to be void or string, but cannot return a string when explicitly marked as `never`.

TypeScript can benefit from the `never` type by performing an exhaustive check. An exhaustive check verifies that every possibility (for all types in the union or all choices in an enum) is handled. The idea is that TypeScript can find an unhandled scenario as early as design-time and also at compilation time. It works by having a potential path that falls under the `else` condition, which returns `never`.

However, when all types of a union or enum cause the code to return something other than `never`, the compiler won't complain. Using `never` is helpful when code around multiple type values evolve. When an option is

added, for example, to a union or enum, TypeScript will compute that the function can return `never` and not compile. Since version 2.[illegible] can find out if the code was entered in the default case (or with `else` case if you are not using the switch statement).

For example, in the code below, there is an `enum` with two items. TypeScript knows that only two cases are possible and the default (`else`) case cannot occur. This insight of TypeScript is perfect since the function return type only accepts a `string` and does not accept `never` . If in the future you add a new item from enum, (for example, a `ChoiceC` , without adding a new case in the switch statement), then, the code can call the `unhandledChoice` function which returns `never` .

```
enum EnumWithChoices {
    ChoiceA,
    ChoiceB,
    ChoiceC,
}

function functionReturnStringFromEnum(c: EnumWithChoices): string {
    switch (c) {
        case EnumWithChoices.ChoiceA:
            return "A";
        case EnumWithChoices.ChoiceB:
            return "B";
        default:
            return unhandledChoiceFromEnum(c);
    }
}

function unhandledChoiceFromEnum(x: never): never {
    throw new Error("Choice not defined");
}
```

The type `never` is also used in the *mapped type* that you will see in later lessons. In every situation where `never` is used, it is to mark that the code should not be in a specific state, else it will not compile.

The primitive type `never` has been around since TypeScript 2.0. Its usage is limited, but its unique characteristics make it powerful. For example, `never` is a subtype of every type but it cannot be a subtype of any type other than itself.

```
function functionReturnNever(): never{
  throw Error("Error Message")
}
let s: string = "A string";
// let n: never = s; // A string is not a subtype of never
let n: never;
try{
  n = functionReturnNever();
  s = n; // Assignable because never is a subtype
}
catch(e){
  console.log(e.message);
}
```

In cases where TypeScript is unable to logically identify a variable as a specific type, it will set the value to `never`. In the following example, the `else` case is theoretically impossible because the `data` variable can only be `number` or `boolean`, however, the `else` is coded anyway. The value of the variable `data` is, in that case, `never`. You can hover on the variable and see this for yourself.

```
declare function ajaxCall(): number | boolean;
let data : number | boolean = ajaxCall();
if (typeof data == "number"){
  console.log(`The data is a number type: ${typeof data}`);
} else if (typeof data == "boolean"){
  console.log(`The data is a boolean type: ${typeof data}`);
} else{
  console.log(`Impossible ELSE case: ${typeof data}`); // Hover data here
}
```

In a few lessons, we will discuss the different types of functions. But while we are still explaining the type `never`, let's take a glimpse at how we define three functions and how they act differently with their inferred type. If you hover your cursor on the variables `a`, `b`, and on the function `c`, you might be surprised to see that the types are `never`, `never`, and `void`. There is a historical reason for this which serves a purpose on how JavaScript is used. Further details will be seen in the `function` lesson.

```
let a = () => {
    throw new Error("A");
}

let b = function() {
    throw new Error("B");
}

function c() {
    throw new Error("C");
}
```

In the end, `never` indicates a state that is not meant to be. An exception is not expected behavior. An infinite loop in a function is not meant to be sustainable in a system, a condition that is never visited should not exist.

Back

Undefined Versus Null

Next

Unknown: A Better any

Completed

Report an Issue