Symbol and Unique Symbol

In this lesson, we see how TypeScript strongly types the primitive type, symbol, and its subtype, unique symbol.

We'll cover the following

- Symbol
- Unique symbol

Symbol#

`Symbol` is a *primitive* type in ECMAScript 2015 and beyond. TypeScript supports the standard. The equal sign assigns a value to a symbol **without** the keyword `new` bu re parentheses, like an object. A symbol's goal is to provide a unique and immutable variable.

A symbol can take a parameter with a string value. Defining two symbols with the same parameter will produce a different symbol. In fact, the parameter is just there to help developers when printing the symbol to the console. It's a way to differentiate them visually.

The main difference between a constant and a symbol is that the symbol is unique. With a string constant, someone could pass a string with the same value as the constant and it would be accepted. However, using a constant symbol, only the same symbol constant would equal that value. Nothing can coerce a symbol into a string. This means that you cannot add a string to it and expect to become a string.

```
let v1 = "value1";
let v2 = "value1";
if (v1 === v2) {
    console.log("Equal when string"); // This will print
}
let s1 = Symbol("value1");
let s2 = Symbol("value1");
if (s1 === s2) {
    console.log("Equal when symbol"); // This will not print, they are not equal
}
```

An object property can be a symbol. Its assignment uses the symbol between brackets. Do keep in mind that a property defined with a symbol won't appear when you invoke `Object.defineProperty` or `Object.getOwnPropertyNames`.

To get all properties defined by symbols, you must use `getOwnPropertySymbols`. If all properties defined are required, you must use `Reflect.ownKeys()`. In the end, the goal is to provide a unique way to define a specific member of the object and avoid a potential collision that a string cannot prevent.

```
const prop1 = Symbol();
const obj = { [prop1]: "p1" };

// console.log(obj.prop1); // Does not compile
console.log(obj[prop1]); // "p1"
```

Unique symbol

A `unique symbol` can only be defined with the use of `const` or `readonly static`. A `unique` symbol is used to create a *literal type* that d[illegible] another symbol. Hence, the type is not `symbol` but a `symbol` with a unique identity.

A way to look at this is the way a `string` can be a `string` or a `string literal`. Hovering over the first variable of the following code shows that it is type `string`, while the type of the second variable is `Value3`.

```
let aString: string = "Value 1";
aString = "Value 2";

const aSecondString = "Value3";
```

Back to the `unique symbol`, it is similar. In the following code, both the first and second variables are of type `Symbol`. However, the last symbol is not of type `Symbol`, but of type `typeof(aThirdSymbol)`.

```
let aSymbol: Symbol = Symbol("Value1");
aSymbol = Symbol("Value2"); // Type is: Symbol

const aSecondSymbol: Symbol = Symbol("Value3");  // Type is: Symbol
const aThirdSymbol: unique symbol = Symbol("Value3");  // Type is: typeof(aThirdSymbol)
```

A `unique symbol` can only be declared with `const`. They are also unique therefore, if compared, will always return `false`. The next example compares a `Symbol` with another `Symbol` as well as to a `unique symbol`.

```
let s1: Symbol = Symbol("s1"); // Type is: Symbol
const s2: Symbol = Symbol("s2");  // Type is: Symbol
const s3: unique symbol = Symbol("s3");  // Type is: typeof(s3)
```

```
const s4: unique symbol = Symbol("s4");  // Type is: typeof(s4)

if (s1 === s2) {
    console.log("S1 and S2 are the same symbol"); // Won't print
}

if (s3 === s2) {
    console.log("S3 and S2 are the same symbol"); // Won't print
}

// if (s3 === s4) {
//     // Does not compile
// }
```

Back

Literal Type to Narrow Primitive Type

Next

Casting to Change Type

Completed

Report an Issue