Type Checking with typeof

This lesson goes over type checking with JavaScript's typeof.

WE'LL COVER THE FOLLOWING

- Borrowing type
- Control flow analysis

Borrowing type

Borrowing the type of another variable is possible by using `typeof`. Borrowing type brings protection at design time by allowing you to set a type without defining an *actual interface or type*. For example, you define a variable with some members using curly braces. There is no actual way to get the schema of the object.

However, with `typeof` you can extract the type from the variable. The extraction is possible for variables or parameters. Instead of a concrete type, the variable uses `typeof` followed by the name of the variable to borrow the schema. This technique is called “type aliases”.

The transposition of the returned type from `typeof` can be in a concrete type by using `type` keyword followed by the name of the type you create by extracting this one from `typeof`.

```
let myObject = { name: "test" };
let myOtherObject: typeof myObject; // Borrow type of the "myObject"
myOtherObject = { name: "test2" };
type TypeFromTypeOf = typeof myOtherObject; // Borrow
```

Control flow analysis

Since **version 2.0**, TypeScript uses, something called **control flow analysis** which allows it to be smart, depending on how the code behaves with some which allows it to be smart, depending on how the code behaves with some variables.

Control flow analysis is especially useful when a variable can be of multiple types; for example, if a variable is defined to be a `number`, a `string` or `undefined`. TypeScript uses the most specific type that a statement can provide to give a hint for the actual underlying type.

For example, if you check for `undefined`, everything after the check will be a `string` or a `number`. **Line 2** in the following code is performing that check. Thus TypeScript knows that `undefined` cannot exist beyond that point and won't complain about checking for `undefined` when accessing members of the variable. In the following example, **line 4 to line 8** are at 100% sure not undefined, hence between `string` or `number` which means we can use `typeof`.

Comparing with `typeof` can narrow down types, if needed. The example was using primitives, but the type can be narrowed down to an object, array, or alias or interface. It's important to note that `typeof` cannot be used against a class or an interface. You will see how to handle more complex scenarios later.

```
function functionCheckTypeOf(param1: number | string | undefined) {
    if (param1 === undefined) { // Check each type, first undefined directly
        console.log("It's undefined");
    } else if (typeof param1 === "number") { // Check primitive with typeof
        console.log("It's a number");
    } else if (typeof param1 === "string") { // Check primitive with typeof
        console.log("It's a string");
    }
}
functionCheckTypeOf(1);
functionCheckTypeOf("1");
functionCheckTypeOf(undefined);
```

`typeof` doesn't work as you may expect with classes and interfaces. In both cases, `typeof` returns `object`. See **line 12** and **13**. To determine the type of class, the use of `instanceOf` is required. Further discussion on `instanceOf` will follow.

For interface, many strategies exist, but it is trickier since there is no runtime equivalence in JavaScript. Interface comparisons will be discussed later.

```
class ClassToUseWithTypeOf {
    private x: string = "val1";
    public y: string = "val2";
}
interface InterfaceTypeOf {
    y: string;
}

const classTypeOf = new ClassToUseWithTypeOf();
const interTypeOf: InterfaceTypeOf = { y: "test" };

console.log("Class TypeOf", typeof classTypeOf); // object
console.log("Interface TypeOf", typeof (interTypeOf)); // object
```

In this lesson, we saw that `typeof` while giving the type remains limited for many scenarios. In any situation an object from a class or an interface or type always ensure to use another type checking pattern described in this course.