TypeScript Scope is JavaScript Scope

This lesson discusses the principle of shadowing, capturing, and declaring a variable in JavaScript and naturally porting to TypeScript.

We'll cover the following

Shadowing scope#

We briefly encountered the concept of scope when exploring the three declaration types discussed in the previous lessons. You saw that a variable declared using `var` has a broader than, `let`, and `const`. However, there are some other cases involving the scope with `let` and `const`.

The first case is *shadowing*. This occurs when one variable is declared twice, in an outer scope, and an inner scope. For example, if you have two loops and both of them are using the variable `i`, TypeScript is smart enough to understand that both declarations are for two different variables. However, it is confusing and susceptible to error, hence it is not recommended even if the code will transpile without a problem.

```
1  function f1(i: number) {
2      console.log("Parameter value", i);
3      let i: number = 10;
4  }
```

```
4  }
5  f1(0);
```

The code above will not allow the parameter and a `let` variable define the same variable name. TypeScript will find and throw an error that the variable has been duplicated. In JavaScript, this error would not have been caught.

The following code demonstrates that it is possible to define the same variable name when the variable is in a different scope. In the example below, each iterative loop has its own scope, hence TypeScript uses the closest `i` depending on the context of execution. This behavior is the same in JavaScript.

```
function f1(i2: number) {
    // i will be shadowed
    console.log("Parameter value:", i2);
    let i: number = 10; // Shadow #1
    console.log("Variable value:", i);
    for (let i = 100; i < 101; i++) {
        // Shadow #2
        console.log("For-statement value:", i);
        for (let i = 200; i < 201; i++) {
            // Shadow #3
            console.log("For-statement value 2:", i);
        }
    }
    console.log("Variable value:", i);
}
f1(0);
```

The following example demonstrates that you can define the same variable name. In the following example, try uncommenting line 4. Suddenly, TypeScript finds a duplicate of declaration in the same scope and throws an

error.

```
function f2() {
    let i: number = 10;
    if(true){
        // console.log("Variable in IF before value:", i);
        let i: number = 100;
        console.log("Variable in IF after value:", i);
    }
    console.log("Variable value:", i);
}
f2();
```

The definition of a variable before it's used is confusing. In the example, the variable `i` is also of type `number` but could have been something else. In the following code, you can see that each scope has its own set of variables. In the previous example, TypeScript finds a variable already defined for the scope.

```
function f3() {
    let i: number = 10; // Shadow #1
    if(true){
        let i: string = "Now a string value";
        console.log("Variable in IF after value:", i);
    }
    console.log("Variable value:", i);
}
f3();
```

Capturing scope

The second case of scoping is called *capturing*. This occurs when you have a variable that you define in an inner scope and then use inside a function that you assign to another scope. During the assignment of the function, the

variable defined in the inner scope will be captured, like a s o

when leaving the scope, the function declared before the inner scope will still have the value of the variable.

```
function mainFunction() {
    let innerFunction;
    if (true) {
        // Environment for capturing start here
        let variableCapturedByTheInnerFunction = "AvailableToTheInnerFunction";
        innerFunction = function() {
            return variableCapturedByTheInnerFunction;
        }; // Environment for capturing stop here
    }
    return innerFunction();
}
console.log(mainFunction());
```

```
function mainFunction2() {
    let innerFunction;
    if (true) {
        // Environment for capturing start here
        let variableCapturedByTheInnerFunction = "AvailableToTheInnerFunction";
        innerFunction = function() {
            return variableCapturedByTheInnerFunction;
        }; // Environment for capturing stop here
        variableCapturedByTheInnerFunction = "Changed"
    }
    return innerFunction();
}
console.log(mainFunction2());
```

The following code does not leverage capturing. The function loops five times from lines 10 to 14 and adds the function to a list. Then, every function is invoked. The result below should be expected.

```
function mainFunction3() {
    let innerFunction;
    let listFunctions = [];
    for (let i = 10; i < 15; i++) {
        innerFunction = function(param1: number) {
            return param1;
        }
        listFunctions.push(innerFunction(i));
    }
    for (let k = 0; k < 5; k++) {
        console.log(listFunctions[k]);
    }
}
mainFunction3();
```

Within a particular scope, you can *freeze* a variable. Lines 5 and 10 create a scope by generating an anonymous function. This function then passes a value that will be available inside the function `f` at any time in the future.

```
function mainFunction2() {
    let innerFunction;
    let listFunctions = [];
    for (let i = 10; i < 15; i++) {

        innerFunction = (function(param1: number) {
            // let f = function() {
            //     return param1;
            // };
            return param1;
        })(i * 100);
        listFunctions.push(innerFunction);
    }
    for (let k = 0; k < 5; k++) {
        console.log(listFunctions[k]);
    }
}
mainFunction2();
```

The result is different from the un-scoped function. Often, with an asynchronous function, you need to ensure that the data passed to the

asynchronous function, you need to ensure that the data passed to the function remains the same, regardless of the future executio e.

Leveraging *capturing* is the JavaScript way to do this. TypeScript allows it by adding type as you will see in a few lessons.

Declaration scope#

Now that we have discussed `var`, `let` and `const`, the difference should be more obvious. One detail that we have not yet explored is the number of declarations that each variable can have under the same scope. While the declaration of `var` is unlimited, declarations with `let` and `const` must be unique per scope.

If you uncomment lines 4 and 6 of the following code, TypeScript notifies you of the duplicated declaration.

```
var x = "First time";
var x = "Second time";
let y = "First time";
// let y = "ERROR!"; // Doesn't compile
const z = "First time";
// const z = "ERROR!"; // Doesn't compile
```

As mentioned, `let` and `const` can be declared once per *scope*. This means you can have several variables with the same name, but only the one in the scope of invocation is visible.

```
let x = 1;
console.log("Outside: " + x)
function myFunction(){
  // console.log("Inside before re-declaring: " + x)
  let x = 2;
  console.log("Inside after re-declaring: " + x)
  if (x == 2) {
```

```
  if (x == 2) {
    let x = 3;
    console.log("Inside IF : " + x)

    function mySecondFunction(){
      // console.log("Inside-Inside before: " + x)
      let x = 4;
      console.log("Inside-Inside after: " + x)
    }
    mySecondFunction();
  }
}
myFunction();
```

The code above has two commented lines of code. Uncommenting the code will result in two errors. TypeScript will figure out that the statement is accessing a variable not defined in the scope.

Back

Hoisting Variables

Next

Switch Scope

Completed

Report an Issue