Theory: Scope of variables

12 minutes 0 / 5 problems solved

We've been declaring all variables with the help of `let` and `const` did not really think twice. Now it's time to learn more about `var` and at the same time get acquainted with `global` and `local` scope of variables.

§1. Local scope

When we create a variable inside a function or block of code, we actually create a **local** variable that is available only within a certain part of code but not in the entire program.

Let's look at the example:

```
function someFunc() {
  let someVar = "local";
  console.log("Some variable in local scope - " + someVar);
}

someFunc();

console.log("Some variable in global scope - " + someVar); // Uncaught ReferenceError: someVar is not defined
```

When we call a function, we can access the variable and display its value on the screen. However, a variable declared inside the function is not available outside of it. Therefore, the variable `someVar` is `local`, in other words, it belongs to the `local scope`.

`Local` variables in their turn can have **function** and **block scope**. A variable declared with `var` is available inside the whole function and has a `function scope`. The variable also may be available just in the block of code, between `{` and `}`; this variable is declared with `let` and has a `block` scope.

We can have several blocks of code in one function:

```
function someFunc2() {
  let funcVar = "function scope variable";
  console.log("Some variable in function local scope - " + funcVar);
  if (funcVar == "function scope variable") {
    let block1Var = "some variable in block local scope";
    console.log(funcVar + 'is available in if block as ' + block1Var);
  } else {
    let block2Var = "some variable in another block of code";

 console.log("In else block is available " + block2Var + " and " + funcVar);

  }

}


someFunc2();
```

As you can see, we can access a variable declared in external function from the internal `if` and `else` blocks of code.

§2. Global scope

A `global` variable is accessible from anywhere in the program, not just a particular block of code. Here is an example:

Current topic:

Scope of variables

Topic depends on:

For loops

Table of contents:

```
let someVar = "global";

function someFunc() {
  console.log("Some variable in local scope - " + someVar);
}

someFunc();
console.log("Some variable in global scope - " + someVar);
```

We will get the following output:

```
Some variable in local scope - global
Some variable in global scope - global
```

The variable `someVar` here is `global`, because it can be called from anywhere in the program, not just in the function where it was declared.

If a variable is declared without a special keyword it's considered a `global` variable by default, no matter where in the program it was declared. For example, try running this in the console:

```
function someFunc() {
  someVar = "global";
  console.log("Some variable in local scope - " + someVar);
}

someFunc();
console.log("Some variable in global scope - " + someVar);
```

You'll see that the output will be like this:

```
Some variable in local scope - global
Some variable in global scope - global
```

§3. Priority of variables

`local` and `global` variables differ in their **priority**: it is higher for `local` variables. Let's consider the following example:

```
someVar = "global";

function someFunc() {
  let someVar = "local";
  console.log("Some variable in local scope - " + someVar);
}

someFunc();
console.log("Some variable in global scope - " + someVar);
```

As a result, we get the following:

```
Some variable in local scope - local
Some variable in global scope - global
```

Even though the variable `someVar` in a `global scope` was declared earlier, when we access the variable `someVar` inside the function, the `local` variable is received because of the priority of `local` variables.

§4. let *Vs* const *Vs* var

As you know, besides `let`, there are two more identifiers for declaring a variable: `const` and `var`.

Unlike it is with the `let` identifier, variables declared with `const` cannot be overridden. See the example below:

```
const someVar = "constant variable";
someVar = "mutable variable";
```

We will get `TypeError`.

Moving on: in earlier versions of JavaScript, it was customary to use `var` to declare a variable. However, today this method is considered obsolete.

As we said above, variable declared with `var` is available inside the whole function and has a so-called `function scope`. For example:

```
function someFunc() {
  var i;
  for (i = 0; i <= 5; i++) {
    var someVar = i * i;
  }
  console.log(i);
  console.log(someVar);
}


someFunc();
```

Here we will get the last values of `i` and `someVar`.

```
function someFunc() {
  let i;
  for (i = 0; i <= 5; i++) {
    let someVar = i * i;
  }
  console.log(i);
  console.log(someVar);
}


someFunc();
```

However, here we will get `ReferenceError` as a result if we try to access `someVar`, because the `let` identifier has a `block scope`, so the variables declared with it are accessible inside the block of code between the `{}` brackets.

§5. Conclusion

In this topic, we examined what the `local` and `global` scopes are, found out that `local` variables can have `function` or `block scope`, looked at different ways of declaring variables, and saw how variables may differ in their scopes.

Report a typo

47 users liked this theory. **2** didn't like it. **What about you?**

Start practicing

Comments (2) Hints (0) Useful links (0) Show discussion