Back to Week 2

JS | Variable scope, hoisting and shadowing

LESSON

Learning goals

After this lesson, you will:

- understand what **scope** and **hoisting** are,
- understand a bit more the new variable declaration approaches in ES6, and what kinds of scopes there are in JavaScript,
- understand what variable shadowing is.

Scope

To work with JavaScript efficiently, one of the first things you need to understand is the concept of `variable scope`.

> The scope of a variable is controlled by the location of the variable declaration, and defines the region of the program where a particular variable is accessible.

One of the most significant changes in ES6 was introducing a new way of declaring variables, using `let` and `const` keywords and these also brought to a play different `scope` than the ones we used to have with `var`.

Kinds of scopes

JavaScript has three scopes – **global scope**, **function scope** and **block scope**.

Let's quickly define each of them:

- A variable with **global level scope** is accessible from anywhere within the script where it was created. It is declared at the topmost level and is accessible from within child scopes, such as functions or statements.
- **Function level scope** is limited to within the function from which it was declared and any child scopes such as nested functions or statements. Variables with the function-level scope are not accessible outside the function they were declared.
- **Block level scope** is limited further to the statement or expression of which the variable was declared. Or in simple terms, anywhere between an opening and closing curly brace `{}`.

`var` and scope

Before the ES6, all variables were declared using `var` keyword. Declaring the variables this way has some specifics.

> Using keyword **`var`** to declare variables, they become available in:
>
> - **global** or
> - in the **function/local** scope.

To simplify, any time a variable is declared outside of a function, it belongs to the **global scope** and can be accessed (used) in the whole window.
If we declare a variable inside the function, then the variable belongs to the **functional or local scope**.

Let’s see it on the example:

```
var message = 'Hello from the global scope!';

function sayHelloFromLocalScope() {
  var greeting = 'Hello from functional/local scope!';
  return greeting;
}

console.log(message); // <== Hello from the global scope!
console.log(greeting); // <== ReferenceError: greeting is not defined
```

Explain this code

As we can see, `message` belongs to the **global scope and can be accessed from anywhere** in the code. In the second example, the `greeting` variable is **functionally/locally scoped and can’t be accessed outside of the function where it was declared**.

To conclude, each function has its own scope, and any variable declared within that function is only accessible from that function and any nested functions. This is called Lexical Scoping.

When we declare a function inside another function, then we create a nested scope.

Now... let’s define a function and try to access the outer variable from that function. We are trying to access the inner variable from the global scope.

Here we can see how inner functions can access outer variables (`outerVar`). However, outer functions can’t access the inner variables (`innerVar`).

```
let outerVar = 1;

function inner() {
  let innerVar = 2;
  console.log(outerVar);
}

console.log(innerVar); // => ReferenceError: innerVar is not defined
```

Explain this code

This *doesn’t apply* to *if statements* and *for loops*. They don’t have their scope (for now).
Let’s see:

```
for (var i = 1; i <= 30; i++) {
  console.log(`Iteration number: ${i}`);
}

console.log(`After the loop: ${i}`);

// [...]
// Iteration number: 28
// Iteration number: 29
// Iteration number: 30
// After the loop 31
```

Explain this code

Using **var** variables **can be re-declared and updated**

The following won't cause the error:

```
var message = 'Hello from the global scope!';
var message = 'This is the second message.';

// OR

var message = 'Hello from the global scope!';
message = 'This is the second message.';
```

Explain this code

But is this necessarily a good thing? Let's see:

```
var name = 'Ironhacker';
if (true) {
  var name = 'Ted';
  console.log(`Name inside if statement: ${name}`);
}

console.log(`Name outside if statement: ${name}`);

// Name inside if statement: Ted
// Name outside if statement: Ted
```

Explain this code

As we can see from the example above, variable *name* is re-declared, and if we didn't know that there's already a variable *name* declared earlier, we could've broken bunch of things.
If we use the same variable throughout our code, and we reassign it like we just did, we won't see the results we expected for sure. That's why `let` and `const` are here to prevent this, and we will see now how.

One more thing to keep in mind - when we run JS in the browser, the outermost scope is the `window` object, where all `var` variables are stored as `window` properties. We call this the **object environment record**.

```
var a = 1;
console.log(window.a === 1); //=> true
```

Explain this code

`let` (and `const`) and scope

Opposite of the *object environment record*, where all variables declared with `var` are stored, variables declared with `let` and `const` are stored in the **declarative environment record**. This meant the next: if variables declared with *let* and *const* are not “trapped” between any curly braces, they also can be accessed from any point in your code, BUT they won’t be attached to the *global Window object* to prevent *global object pollution*:

```
const a = 1;
console.log(window.a); // undefined
```

Explain this code

In general, `let` and `const` came to fix the issues `var` have had.

Reminder: We can say that **block is any code between open and closed curly braces `{}`**.

> **`let`** gives us **`block scoping`**, is *not* bounded to the `global` or `window` object by default, and should be used in favor of `var`.

If we replace `var` with `let` in one of the previous examples, we will get a different result.

```
for (let i = 1; i <= 30; i++) {
  console.log(`Iteration number: ${i}`);
}

console.log(`After the loop: ${i}`);

// [...]
// Iteration number: 29
// Iteration number: 30
// Iteration number: 30
//
// console.log("After the loop", i);
//                                ^
// ReferenceError: i is not defined
```

Explain this code

👍 Sometimes *errors are ok*. `let` can help us prevent some JavaScript pitfalls, by throwing an error back at us.

As we can see, `let` gives us the **block** scope, which means that the variable declared in a block using *let* can only be used in that block. Blocks also include *if* statements and *for* loops as well as functions.

> Using **let** variables **can’t be re-declared but can be updated**.

Let’s see:

```
// THIS IS OKAY
let message = 'This is the first message.';
message = 'This is the second message.'; // <== This is the second message.
```

Explain this code

```
// BUT THIS WILL THROW AN ERROR

let message = 'This is the first message.';
let message = 'This is the second message.'; // <== Duplicate declaration "message"
```

Explain this code

However, if the same-named variables belong to different scopes, no error will be shown because they are treated as different variables that belong to different scopes.

```
let name = 'Ironhacker';

if (true) {
  let name = 'Ted';
  console.log(`Name inside if statement: ${name}`);
}

console.log(`Name outside if statement: ${name}`);

// Name inside if statement: Ted
// Name outside if statement: Ironhacker
```

Explain this code

To recap and compare: `let` gives us much more security because when variables are declared with `let`, if declared in different scopes, are two different variables while using `var` the second one will re-declare the first one. At the same time, `let` doesn't allow having the same-named variables in the same scope while, as we saw, with `var` that is possible to happen, and no error will be thrown.

> Using **const** variables **can't be re-declared nor updated**.

The most secure and preferred way of declaring variables is using `const` **- variables declared this way couldn't be re-declared nor updated**.

```js
// THIS WILL THROW AN ERROR
const message = 'This is the first message.';
message = 'This is the second message.'; // <== "message" is read-only

// AND THIS WILL THROW AN ERROR

const message = 'This is the first message.';
const message = 'This is the second message.'; // <== Duplicate declaration "message"
```

Explain this code

Variables declared with `const` have to be initialized in the moment of declaration.

```js
const name = "John"; // <== CORRECT

const name; // <== WRONG!
name = "John"; // <== this doesn't work
```

Explain this code

We saw in the case of objects and arrays declared using *const*, we can update the existing properties.

A quick reminder:

In the case of declaring an object using the `const` keyword, this means that new properties and values can be added. Still, the value of the object itself is fixed to the same reference (address) in the memory and the object (or any variable declared with const) can’t be reassigned.

```js
// This is ok
const obj = {};
obj.name = 'Ironhacker';

// This is not ok
obj = { name: 'Ironhacker' };
// SyntaxError: Assignment to constant variable
```

Explain this code

Hoisting

Hoisting is a JavaScript mechanism where **variables and function declarations are moved to the top of their scope before code execution**.

This means you can use the variable in the parts of your code before you declared it officially.

We touched upon this concept in the Functions Advanced lesson, so check it out to refresh your memory.

`var` and hoisting

Variables declared using `var` are moved to the top if it's scope (we say - hoisted) and initialized with a value of `undefined`.

```
console.log(message); // => undefined
var message = 'Hello from the global scope!';
```

Explain this code

`let` (and `const`) hoisting

There's a slight difference between `var` and `let` when it comes to hoisting: variables declared with *let* are hoisted to the top as well, but they are not initialized. So using *var* we would get the value of *undefined*, but using *let* we get a *Reference Error*.
According to the official docs, the reason for that is:

> `let` and `const` hoist, but you can't access them before the actual declaration is evaluated at runtime.

```
console.log(message); // => ReferenceError: Cannot access 'message' before initialization
let message = 'Hello from the global scope!';
```

Explain this code

Shadowing

A variable shadowing occurs when a variable declared within a certain scope has the same name as a variable declared in an outer scope.

Let's check the following example:

```
let a = 1;
let b = 2;

function inner() {
  a = 4; // reassigned
  let b = 3; // declared in an inner scope
}

inner();
console.log(a); // => 4
console.log(b); // => 2
```

Explain this code

> **Note**: In **strict mode** (`"use strict"`), you get an error if you assign a value to a variable without first declaring the variable. Perhaps a good idea for those who tend to forget the declaration phase.

When JavaScript sees a reference to a variable, it will try to find the variable declaration within the same scope where it has been referenced. If it can't find that declaration, it will look for it within the parent scope. If it can't find it there, it will look for the grand-parent scope... and will keep trying until it reaches the `global scope`.

Extra

💡 *Potential interview questions*:

	hoisted/initialized	scope	updated	re-declared
var	✅/✅	global or functional(local)	✅	✅
let	✅/❌	block	✅	❌
const	✅/❌	block	❌	❌

⭐ `var` and `let` can be declared and later on initialized.

⭐ `const` have to be initialized when declared.

Summary

A **scope** represents where a declared variable is available to be used, or better saying - accessed.

In this lesson, we learned about the new features of ES6, caused by introducing `const` and `let`, and we learned about similarities and differences between declaring variables using `var` vs. `let` and `const` in a matter of the scope. When talking about the scope, we again touched upon the concept of hoisting. Also, we learned what a variable shadowing is.

Extra Resources

- MDN - Introduction to variable scope
- MDN `const`
- MDN `let`
- Variable shadowing

Mark as completed

PREVIOUS
← Bonus: LAB | CSS Flexbox Slack

NEXT
JS | Debugging, Error Handling and JS Hint →