Scope

**Slide 1.1**

Scope - is a logical boundaries, that determines the accessibility of variables, objects, and functions from different parts of the code. A scope in JavaScript defines what variables you have access to.

**Slide 1.2**

In JavaScriptes there are two kinds of scope – global scope and local scope.

**Slide 2.1**

If a variable is declared outside all functions or curly braces ({}), it is said to be defined in the **global scope**.

All scripts and functions on a web page can access it. Once you've declared a global variable, you can use this variable anywhere in your code, even in functions.

var global = 'I am global variable!'

function saySomething() {

console.log(global)

}

console.log(global) // 'I am global variable!'

saySomething() // 'I am global variable!'

In the code we defined a variable with name global and assigned it a value. We access our variable when we try to display it with the help of console.log function, another place where there is an access to the variable is in saySomething function. Therefore calling console.log and saySomething function will return the same result.

**Slide 2.2**

Although you can declare variables in the global scope, it is advised not to. This is because there is a chance of naming collisions, where two or more variables are named the same.

If you declare your variables with var, your second variable overwrites the first one after it is declared. This undesirable as you make your code hard to debug. In this example console.log function will return value ‘something else’.

var thing = 'something'

var thing = 'something else'

console.log(thing) // 'something else'

If you declared your variables with const or let, you would receive an error whenever a name collision happens. This is undesirable too.

// Don't do this!

let thing = 'something'

let thing = 'something else' // Error, thing has already been declared

**Slide 2.3**

**Automatically Global**

If you assign a value to a variable that has not been declared, it will automatically become a **global** variable.

This code example will declare a global variable **animal**, even if the value is assigned inside a function, so you can use this variable anywhere in your code.

myFunction();  
// code here can use animal  
function myFunction() {  
    animal = "Cat";  
}

## **Global Variables in HTML**

## In HTML, the global scope is the window object. All global variables belong to this object. After declaring a variable animal, it can be accessed through window.animal

**Slide 3.1**

**Local Scope**

Variables that are usable only in a specific part of your code are considered to be in a local scope. These variables are also called **local variables**.

In JavaScript, there are two kinds of local scope: function scope and block scope(ES2015+).

Let's talk about function scopes first.

**Slide 3.2**

**Function scope**

When you declare a variable in a function, you can access this variable only within the function. You can't get this variable once you get out of it. Variables declared within a JavaScript function are local variables and have local scope. Function parameters also count as local variables and are defined only within the body of the function.

Let’s look at an example. We’ll create a function and declare the variable animal within that function. Animal is accessible and can be used anywhere within that function. However, calling this variable outside of the function will result in an **Uncaught ReferenceError**, because this variable isn’t defined outside this function.

function localScopeExample(){  
 // LOCAL SCOPE  
 var animal = 'Cat';  
 console.log(animal); // Cat  
}

// GLOBAL SCOPE  
console.log(animal); // Uncaught ReferenceError: animal is not defined

**Slide 3.3**

Since local variables are only accessible within their functions, you can use the same variable name in different functions:

In the example we create two functions inside which we define variable animal. There is no collision because we create two variables and each of them is defined in its scope.

function firstFunction(){  
 var animal = 'Cat';  
 console.log(animal); // Cat  
}

function secondFunction(){  
 var animal = 'Dog';  
 console.log(animal); // Dog  
}

But functions don’t have access to each other's scopes when you define them separately, even though one function may be used in another. So if we try to output variable that has been declared in first function, we’ll receive reference error, because this variable is not defined.

**Siled 3.4**

**Block scope**

With var, a variable is either globally scoped, or locally scoped to the function in which it is defined. Block scopes like if, for, while, {}, etc. have no effect on var.

let & const, on the other hand, are scoped within the block which they are defined. When you declare a variable with const or let within a curly brace ({}), you can access this variable only within that curly brace. The block scope is a subset of a function scope since functions need to be declared with curly braces (unless you're using [arrow functions](https://zellwk.com/blog/es6/#arrow-functions) with an implicit return).

Let’s look at an example. You can see that localVariable is scoped to the curly brace:

{

const localVariable = 'I am local variable'

console.log(localVariable) // 'I am local variable'

}

console.log(localVariable) // Error, localVariable is not defined

**Slide 4.1**

**Nested scopes**

Function definitions can be nested. When a function is defined in another function, the inner function has access to the outer function's variables. This behavior is called **lexical scoping**. Lexical scope is the ability of an inner function to access the scope of an outer function. However, the outer function does not have access to the inner function's variables.

function outerFunction () {

const outer = `I'm the outer function!`

function innerFunction() {

const inner = `I'm the inner function!`

console.log(outer) // I'm the outer function!

}

console.log(inner) // Error, inner is not defined

}

In the example while trying to display variable outer in function innerFunction, we try to find this variable in scope of inner function but there isn’t such variable, that’s why we go to the scope above(external scope for the scope where we are now) and try to find our variable there. It’s all right and we find there variable outer with value `I'm the outer function!` and display it. But this rule doesn’t work vice versa. While trying to display variable inner in outer function we get an error because this variable isn’t declared neither in outerFunction scope, nor in global scope

**Slide 5.1**

**Scope shadowing**

Within the body of a function, a local variable takes precedence over a global variable with the same name. If you declare a local variable or function parameter with the same name as a global variable, you effectively hide the global variable:

var scope = "global"; *// Declare a global variable*

function checkscope() {

var scope = "local"; *// Declare a local variable with the same name*

return scope; *// Return the local value, not the global one*

}

checkscope() *// "local"*

**Slide 5.1**

**Hoisting**

Hoisting is JavaScript's default behavior of moving declarations to the top. So in JavaScript, a variable can be declared after it has been used. In other words; a variable can be used before it has been declared.

**Slide 5.2**

Functions, when declared with a function declaration, are always hoisted to the top of the current scope. So, these two are equivalent:

saySomething()

function saySomething () {

console.log('Hello!')

}

function saySomething () {

console.log('Hello!')

}

saySomething()

When declared with a function expression, functions are not hoisted to the top of the current scope.

saySomething() // Error, saySomething is not defined

const saySomething = function () {

console.log('Hello!');

}

Because of these two variations, function hoisting can potentially be confusing, and should not be used. Always declare your functions before you use them.

**Slide 5.3**

var scope = "global";

function f() {

console.log(scope); *// Prints "undefined", not "global"*

var scope = "local"; *// Variable initialized here, but defined everywhere*

console.log(scope); *// Prints "local"*

}

You might think that the first line of the function would print “global”, because the var statement declaring the local variable has not yet been executed. Because of the rules of function scope, however, this is not what happens. The local variable is defined throughout the body of the function, which means the global variable by the same name is hidden throughout the function.

But the local variable is not initialized until the var statement is executed. This is because only the declaration, not the initialization is hoisted to the top.

The function is equivalent to the following. Because of hoisting, scope has been declared before it is used, but because initializations are not hoisted, the value of scope is undefined.

**function** f() {

**var** scope;

console.log(scope);

scope = "local";

console.log(scope);

}

To avoid bugs, always declare all variables at the beginning of every scope.

**Slide 5.4**

if (('magic' in window) === false) {

var magic = 'Something happened' ;

}

console.log(magic);

One more consequence of hoisting. At first it might seem that magic will be defined as 'Something happen'. But because of hoisting the code above will be rewritten in such a way:

var magic

if (('magic' in window) === false) {

magic = 'Something happened';

}

console.log(magic);

The variable will be declared before the condition is checked so it will be put in window and this condition will always be false, therefore variable magic will always be undefined

Variables defined with **let** are not hoisted to the top. Using a **let** variable before it is declared will result in a ReferenceError.

**Slide 6.1**

Do NOT create global variables unless you intend to. Any function, including the window object, can overwrite your global variables and functions.

Use local variables because of

* **Security** — variables are only accessible where they are needed.
* **Reducing Namespace Collisions** — Namespace collisions happen when two or more variables share a common name.

**Slide 6.2**

Here is a small reminder about the ways of declaring of variables depending on the scope. For example, with the help of var we can declare a function both in local and global scopes, but such declaration can’t be used in block scope, also variables declared using var can be redefined, unlike variables declared using const. Attempt to override will cause an error.

