Hello, today I will tel about Js Scope and Closures

Understanding “scope” and “closures” is key to learning JavaScript

Global are variables and functions that are not inside a function. That is, in other words, if a variable or function is not inside a function construct, then they are “global”.

In JavaScript, all global variables and functions are properties of a special object called a global object.

Assigning or reading a global variable, we actually work with the window properties.

You can create a variable by explicit assignment in window:

The execution of the script occurs in two phases:

1. The first phase is initialization, preparation for launch.

During initialization, the script is scanned for the declaration of functions of the type Function Declaration, and then for the declaration of var variables. Each such ad is added to the window.

Functions declared as Function Declaration are created immediately and variables are set to undefined.

2. In the second phase - actually, implementation.

The assignment (=) of variable values occurs when the execution thread reaches the corresponding line of code, until then they are undefined.

Constructions for, if ... do not affect the visibility of variables

Curly brackets, which are used in for, while, if, unlike function declarations, have a “decorative” character.

In JavaScript, there is no difference between a declaration outside the block:

There is also no difference between a declaration in a loop and outside it:

In both cases, the variable will be created before the loop is executed, at the initialization stage, and its value will be saved after the end of the loop.

It doesn't matter where and how many times the variable is declared.

Var ads can be any number of.

All var will be processed once, during the initialization phase.

During the execution phase, var declarations will be ignored: they have already been processed. But assignments will be made.

As a result of initialization, to the beginning of the code execution:

1. Functions declared as Function Declaration are completely created and ready to use.

2. Variables declared, but equal to undefined. Assignments will be executed later when execution reaches them.

And, now, I will tell about Closures and internal functions

All variables inside the function are properties of a special internal LexicalEnvironment object that is created when it is launched.

We will call this object “lexical environment” or simply “variable object”.

At startup, the function creates a LexicalEnvironment object, writes there arguments, functions, and variables. The initialization process is performed in the same order as for the global object, which, generally speaking, is a special case of the lexical environment.

Unlike window, the LexicalEnvironment object is internal, it is hidden from direct access.

Let's take an example to better understand how this works:

When calling a function:

1. Before executing the first line of its code, at the initialization stage, the interpreter creates an empty LexicalEnvironment object and fills it.

In this case, the name argument and the only variable phrase get there:

2. The function is executed.

At run time, the local variable phrase is assigned, that is, in other words, the new value is assigned to the phrase property in LexicalEnvironment object.

At the end of the function the object with variables is usually thrown away and the memory is cleared. In the examples above, this is what happens. After some time, we will consider more complex situations in which an object with variables is preserved even after the function ends.

From the function, we can access not only a local variable, but also an external one:

When accessing a variable, the interpreter first tries to find a variable in the current LexicalEnvironment, and then, if it is not there, it searches for an external variable object. In this case, it is a window.

This search order is possible due to the fact that the reference to the external object of variables is stored in a special internal property of the function, which is called [[Scope]]. This property is closed from direct access, but knowledge of it is very important for understanding how JavaScript works.

When created, the function gets the hidden [[Scope]] property, which refers to the lexical environment in which it was created.

In the example , such an environment is window, so a property is created sayHi

This property never changes. It follows function everywhere, thus binding it to the place of its birth.

When a function is launched, its variable object LexicalEnvironment receives a reference to an “external lexical environment” with a value from [[Scope]].

If the variable is not found in the function, it will be searched outside.

It is thanks to this mechanics in the example above that alert (userName) outputs an external variable. At the code level, this looks like a search in the outer scope, outside the function.

Each function upon creation receives a [[Scope]] reference to an object with variables in the context of which it was created.

When the function starts, a new object is created with LexicalEnvironment variables. It receives a reference to an external variable object from [[Scope]].

When searching for variables, it is carried out first in the current variable object, and then on this link.

It looks so simple that it is not clear - why bother to talk about it at all [[Scope]], about variable objects. They would say: "The function reads variables from the outside" - and that's all. But knowledge of these details will allow us to easily explain and understand more complex situations that we will encounter next.

The value of a variable from the outer region is always current. It may not be what it was at the time the function was created.

A closure is a function along with all external variables that are accessible to it.

This is the standard definition found in Wikipedia and most serious programming sources. That is, a closure is a function + external variables.

However, there is a small terminological feature in JavaScript.

Usually, by saying “closing the function”, they do not mean this function itself, but external variables.

Sometimes they say "the variable is taken from the circuit." This means from an external variable object.

“Understand closures” in JavaScript means to understand the following things:

1. All variables and parameters of functions are properties of the variable object LexicalEnvironment. Each function launch creates a new such object. At the top level, it is the “global object”, in the browser - the window.

2. When created, the function gets the [[Scope]] system property, which refers to the LexicalEnvironment in which it was created.

3. When calling a function, wherever it is passed in the code, it will look for variables first in itself and then in external LexicalEnvironment from the place of its “birth”.

And Finally. Variables: let and const and block scope

Declarations of a variable through let and const have three main differences from var:

The scope of the let and const variables is the {...} block.

As we remember, the variable declared via var is visible everywhere in the function.

The variable declared via let is visible only within the block {...} in which it is declared.

This affects declarations within if, while, or for.

In the example , apples are one variable for all code that is modified in if.

The same with let will work differently:

Here, two independent variables apples, one - global, the second - in the if block.

Note that if the declaration of let apples are deleted in the first line (\*), then the last alert will contain an error: the variable is not defined.

This is because the let and const variables is always visible in the block where it is declared.

It is all , thank for your attention