**BELARUSIAN STATE UNIVERSITY OF INFORMATICS AND RADIOELECTRONICS**

**The English Language**

**Translation**

Master student Kozyakov A. I.

Group 815042

Minsk 2018

Source: https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/README.md#you-dont-know-js-scope--closures

**You Don’t Know JS** by Kyle Simpson

**Preface**

I'm sure you noticed, but "JS" in the book series title is not an abbreviation for words used to curse about JavaScript, though cursing at the language's quirks is something we can probably all identify with!

From the earliest days of the web, JavaScript has been a foundational technology that drives interactive experience around the content we consume. While flickering mouse trails and annoying pop-up prompts may be where JavaScript started, nearly 2 decades later, the technology and capability of JavaScript has grown many orders of magnitude, and few doubt its importance at the heart of the world's most widely available software platform: the web.

But as a language, it has perpetually been a target for a great deal of criticism, owing partly to its heritage but even more to its design philosophy. Even the name evokes, as Brendan Eich once put it, "dumb kid brother" status next to its more mature older brother "Java". But the name is merely an accident of politics and marketing. The two languages are vastly different in many important ways. "JavaScript" is as related to "Java" as "Carnival" is to "Car".

Because JavaScript borrows concepts and syntax idioms from several languages, including proud C-style procedural roots as well as subtle, less obvious Scheme/Lisp-style functional roots, it is exceedingly approachable to a broad audience of developers, even those with just little to no programming experience. The "Hello World" of JavaScript is so simple that the language is inviting and easy to get comfortable with in early exposure.

While JavaScript is perhaps one of the easiest languages to get up and running with, its eccentricities make solid mastery of the language a vastly less common occurrence than in many other languages. Where it takes a pretty in-depth knowledge of a language like C or C++ to write a full-scale program, full-scale production JavaScript can, and often does, barely scratch the surface of what the language can do.

Sophisticated concepts which are deeply rooted into the language tend instead to surface themselves in *seemingly* simplistic ways, such as passing around functions as callbacks, which encourages the JavaScript developer to just use the language as-is and not worry too much about what's going on under the hood.

It is simultaneously a simple, easy-to-use language that has broad appeal, and a complex and nuanced collection of language mechanics which without careful study will elude *true understanding* even for the most seasoned of JavaScript developers.

Therein lies the paradox of JavaScript, the Achilles' Heel of the language, the challenge we are presently addressing. Because JavaScript *can* be used without understanding, the understanding of the language is often never attained.

**Mission**

If at every point that you encounter a surprise or frustration in JavaScript, your response is to add it to the blacklist, as some are accustomed to doing, you soon will be relegated to a hollow shell of the richness of JavaScript.

While this subset has been famously dubbed "The Good Parts", I would implore you, dear reader, to instead consider it the "The Easy Parts", "The Safe Parts", or even "The Incomplete Parts".

This *You Don't Know JavaScript* book series offers a contrary challenge: learn and deeply understand *all* of JavaScript, even and especially "The Tough Parts".

Here, we address head on the tendency of JS developers to learn "just enough" to get by, without ever forcing themselves to learn exactly how and why the language behaves the way it does. Furthermore, we eschew the common advice to *retreat*when the road gets rough.

I am not content, nor should you be, at stopping once something *just works*, and not really knowing *why*. I gently challenge you to journey down that bumpy "road less traveled" and embrace all that JavaScript is and can do. With that knowledge, no technique, no framework, no popular buzzword acronym of the week, will be beyond your understanding.

These books each take on specific core parts of the language which are most commonly misunderstood or under-understood, and dive very deep and exhaustively into them. You should come away from reading with a firm confidence in your understanding, not just of the theoretical, but the practical "what you need to know" bits.

The JavaScript you know *right now* is probably *parts* handed down to you by others who've been burned by incomplete understanding. *That* JavaScript is but a shadow of the true language. You don't *really* know JavaScript, *yet*, but if you dig into this series, you *will*. Read on, my friends. JavaScript awaits you.

**Summary**

JavaScript is awesome. It's easy to learn partially, and much harder to learn completely (or even *sufficiently*). When developers encounter confusion, they usually blame the language instead of their lack of understanding. These books aim to fix that, inspiring a strong appreciation for the language you can now, and *should*, deeply *know*.

Note: Many of the examples in this book assume modern (and future-reaching) JavaScript engine environments, such as ES6. Some code may not work as described if run in older (pre-ES6) engines.

# Chapter 1: What is Scope?

One of the most fundamental paradigms of nearly all programming languages is the ability to store values in variables, and later retrieve or modify those values. In fact, the ability to store values and pull values out of variables is what gives a program state.

Without such a concept, a program could perform some tasks, but they would be extremely limited and not terribly interesting.

But the inclusion of variables into our program begets the most interesting questions we will now address: where do those variables live? In other words, where are they stored? And, most importantly, how does our program find them when it needs them?

These questions speak to the need for a well-defined set of rules for storing variables in some location, and for finding those variables at a later time. We'll call that set of rules: Scope.

But, where and how do these Scope rules get set?

## Compiler Theory

It may be self-evident, or it may be surprising, depending on your level of interaction with various languages, but despite the fact that JavaScript falls under the general category of "dynamic" or "interpreted" languages, it is in fact a compiled language. It is not compiled well in advance, as are many traditionally-compiled languages, nor are the results of compilation portable among various distributed systems.

But, nevertheless, the JavaScript engine performs many of the same steps, albeit in more sophisticated ways than we may commonly be aware, of any traditional language-compiler.

In traditional compiled-language process, a chunk of source code, your program, will undergo typically three steps before it is executed, roughly called "compilation":

1. **Tokenizing/Lexing:** breaking up a string of characters into meaningful (to the language) chunks, called tokens. For instance, consider the program: var a = 2;. This program would likely be broken up into the following tokens: var, a, =, 2, and ;. Whitespace may or may not be persisted as a token, depending on whether it's meaningful or not.

**Note:** The difference between tokenizing and lexing is subtle and academic, but it centers on whether or not these tokens are identified in a stateless or stateful way. Put simply, if the tokenizer were to invoke stateful parsing rules to figure out whether a should be considered a distinct token or just part of another token, that would be **lexing**.

1. **Parsing:** taking a stream (array) of tokens and turning it into a tree of nested elements, which collectively represent the grammatical structure of the program. This tree is called an "AST" (**A**bstract **S**yntax **T**ree).

The tree for var a = 2; might start with a top-level node called VariableDeclaration, with a child node called Identifier (whose value is a), and another child called AssignmentExpression which itself has a child called NumericLiteral (whose value is 2).

1. **Code-Generation:** the process of taking an AST and turning it into executable code. This part varies greatly depending on the language, the platform it's targeting, etc.

So, rather than get mired in details, we'll just handwave and say that there's a way to take our above described AST for var a = 2; and turn it into a set of machine instructions to actually create a variable called a (including reserving memory, etc.), and then store a value into a.

**Note:** The details of how the engine manages system resources are deeper than we will dig, so we'll just take it for granted that the engine is able to create and store variables as needed.

The JavaScript engine is vastly more complex than just those three steps, as are most other language compilers. For instance, in the process of parsing and code-generation, there are certainly steps to optimize the performance of the execution, including collapsing redundant elements, etc.

So, I'm painting only with broad strokes here. But I think you'll see shortly why these details we do cover, even at a high level, are relevant.

For one thing, JavaScript engines don't get the luxury (like other language compilers) of having plenty of time to optimize, because JavaScript compilation doesn't happen in a build step ahead of time, as with other languages.

For JavaScript, the compilation that occurs happens, in many cases, mere microseconds (or less!) before the code is executed. To ensure the fastest performance, JS engines use all kinds of tricks (like JITs, which lazy compile and even hot re-compile, etc.) which are well beyond the "scope" of our discussion here.

Let's just say, for simplicity's sake, that any snippet of JavaScript has to be compiled before (usually right before!) it's executed. So, the JS compiler will take the program var a = 2; and compile it first, and then be ready to execute it, usually right away.

## Understanding Scope

The way we will approach learning about scope is to think of the process in terms of a conversation. But, who is having the conversation?

### The Cast

Let's meet the cast of characters that interact to process the program var a = 2;, so we understand their conversations that we'll listen in on shortly:

1. Engine: responsible for start-to-finish compilation and execution of our JavaScript program.
2. Compiler: one of Engine's friends; handles all the dirty work of parsing and code-generation (see previous section).
3. Scope: another friend of Engine; collects and maintains a look-up list of all the declared identifiers (variables), and enforces a strict set of rules as to how these are accessible to currently executing code.

For you to fully understand how JavaScript works, you need to begin to think like Engine (and friends) think, ask the questions they ask, and answer those questions the same.

### Back & Forth

When you see the program var a = 2;, you most likely think of that as one statement. But that's not how our new friend Engine sees it. In fact, Engine sees two distinct statements, one which Compiler will handle during compilation, and one which Engine will handle during execution.

So, let's break down how Engine and friends will approach the program var a = 2;.

The first thing Compiler will do with this program is perform lexing to break it down into tokens, which it will then parse into a tree. But when Compiler gets to code-generation, it will treat this program somewhat differently than perhaps assumed.

A reasonable assumption would be that Compiler will produce code that could be summed up by this pseudo-code: "Allocate memory for a variable, label it a, then stick the value 2 into that variable." Unfortunately, that's not quite accurate.

Compiler will instead proceed as:

1. Encountering var a, Compiler asks Scope to see if a variable a already exists for that particular scope collection. If so, Compiler ignores this declaration and moves on. Otherwise, Compiler asks Scope to declare a new variable called a for that scope collection.
2. Compiler then produces code for Engine to later execute, to handle the a = 2 assignment. The code Engine runs will first ask Scope if there is a variable called a accessible in the current scope collection. If so, Engine uses that variable. If not, Engine looks elsewhere (see nested Scope section below).

If Engine eventually finds a variable, it assigns the value 2 to it. If not, Engine will raise its hand and yell out an error!

To summarize: two distinct actions are taken for a variable assignment: First, Compiler declares a variable (if not previously declared in the current scope), and second, when executing, Engine looks up the variable in Scope and assigns to it, if found.

### Compiler Speak

We need a little bit more compiler terminology to proceed further with understanding.

When Engine executes the code that Compiler produced for step (2), it has to look-up the variable a to see if it has been declared, and this look-up is consulting Scope. But the type of look-up Engine performs affects the outcome of the look-up.

In our case, it is said that Engine would be performing an "LHS" look-up for the variable a. The other type of look-up is called "RHS".

I bet you can guess what the "L" and "R" mean. These terms stand for "Left-hand Side" and "Right-hand Side".

Side... of what? **Of an assignment operation.**

In other words, an LHS look-up is done when a variable appears on the left-hand side of an assignment operation, and an RHS look-up is done when a variable appears on the right-hand side of an assignment operation.

Actually, let's be a little more precise. An RHS look-up is indistinguishable, for our purposes, from simply a look-up of the value of some variable, whereas the LHS look-up is trying to find the variable container itself, so that it can assign. In this way, RHS doesn't really mean "right-hand side of an assignment" per se, it just, more accurately, means "not left-hand side".

Being slightly glib for a moment, you could also think "RHS" instead means "retrieve his/her source (value)", implying that RHS means "go get the value of...".

Let's dig into that deeper.

When I say:

console.log( a );

The reference to a is an RHS reference, because nothing is being assigned to a here. Instead, we're looking-up to retrieve the value of a, so that the value can be passed to console.log(..).

By contrast:

a = 2;

The reference to a here is an LHS reference, because we don't actually care what the current value is, we simply want to find the variable as a target for the = 2 assignment operation.

**Note:** LHS and RHS meaning "left/right-hand side of an assignment" doesn't necessarily literally mean "left/right side of the = assignment operator". There are several other ways that assignments happen, and so it's better to conceptually think about it as: "who's the target of the assignment (LHS)" and "who's the source of the assignment (RHS)".

Consider this program, which has both LHS and RHS references:

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

The last line that invokes foo(..) as a function call requires an RHS reference to foo, meaning, "go look-up the value of foo, and give it to me." Moreover, (..) means the value of foo should be executed, so it'd better actually be a function!

There's a subtle but important assignment here. **Did you spot it?**

You may have missed the implied a = 2 in this code snippet. It happens when the value 2 is passed as an argument to the foo(..) function, in which case the 2 value is **assigned** to the parameter a. To (implicitly) assign to parameter a, an LHS look-up is performed.

There's also an RHS reference for the value of a, and that resulting value is passed to console.log(..). console.log(..)needs a reference to execute. It's an RHS look-up for the console object, then a property-resolution occurs to see if it has a method called log.

Finally, we can conceptualize that there's an LHS/RHS exchange of passing the value 2 (by way of variable a's RHS look-up) into log(..). Inside of the native implementation of log(..), we can assume it has parameters, the first of which (perhaps called arg1) has an LHS reference look-up, before assigning 2 to it.

**Note:** You might be tempted to conceptualize the function declaration function foo(a) {... as a normal variable declaration and assignment, such as var foo and foo = function(a){.... In so doing, it would be tempting to think of this function declaration as involving an LHS look-up.

However, the subtle but important difference is that Compiler handles both the declaration and the value definition during code-generation, such that when Engine is executing code, there's no processing necessary to "assign" a function value to foo. Thus, it's not really appropriate to think of a function declaration as an LHS look-up assignment in the way we're discussing them here.

### Engine/Scope Conversation

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

Let's imagine the above exchange (which processes this code snippet) as a conversation. The conversation would go a little something like this:

***Engine***: Hey Scope, I have an RHS reference for foo. Ever heard of it?

***Scope***: Why yes, I have. Compiler declared it just a second ago. He's a function. Here you go.

***Engine***: Great, thanks! OK, I'm executing foo.

***Engine***: Hey, Scope, I've got an LHS reference for a, ever heard of it?

***Scope***: Why yes, I have. Compiler declared it as a formal parameter to foo just recently. Here you go.

***Engine***: Helpful as always, Scope. Thanks again. Now, time to assign 2 to a.

***Engine***: Hey, Scope, sorry to bother you again. I need an RHS look-up for console. Ever heard of it?

***Scope***: No problem, Engine, this is what I do all day. Yes, I've got console. He's built-in. Here ya go.

***Engine***: Perfect. Looking up log(..). OK, great, it's a function.

***Engine***: Yo, Scope. Can you help me out with an RHS reference to a. I think I remember it, but just want to double-check.

***Scope***: You're right, Engine. Same guy, hasn't changed. Here ya go.

***Engine***: Cool. Passing the value of a, which is 2, into log(..).

...

### Quiz

Check your understanding so far. Make sure to play the part of Engine and have a "conversation" with the Scope:

function foo(a) {

var b = a;

return a + b;

}

var c = foo( 2 );

1. Identify all the LHS look-ups (there are 3!).
2. Identify all the RHS look-ups (there are 4!).

**Note:** See the chapter review for the quiz answers!

## Nested Scope

We said that Scope is a set of rules for looking up variables by their identifier name. There's usually more than one Scope to consider, however.

Just as a block or function is nested inside another block or function, scopes are nested inside other scopes. So, if a variable cannot be found in the immediate scope, Engine consults the next outer containing scope, continuing until found or until the outermost (aka, global) scope has been reached.

Consider:

function foo(a) {

console.log( a + b );

}

var b = 2;

foo( 2 ); // 4

The RHS reference for b cannot be resolved inside the function foo, but it can be resolved in the Scope surrounding it (in this case, the global).

So, revisiting the conversations between Engine and Scope, we'd overhear:

***Engine***: "Hey, Scope of foo, ever heard of b? Got an RHS reference for it."

***Scope***: "Nope, never heard of it. Go fish."

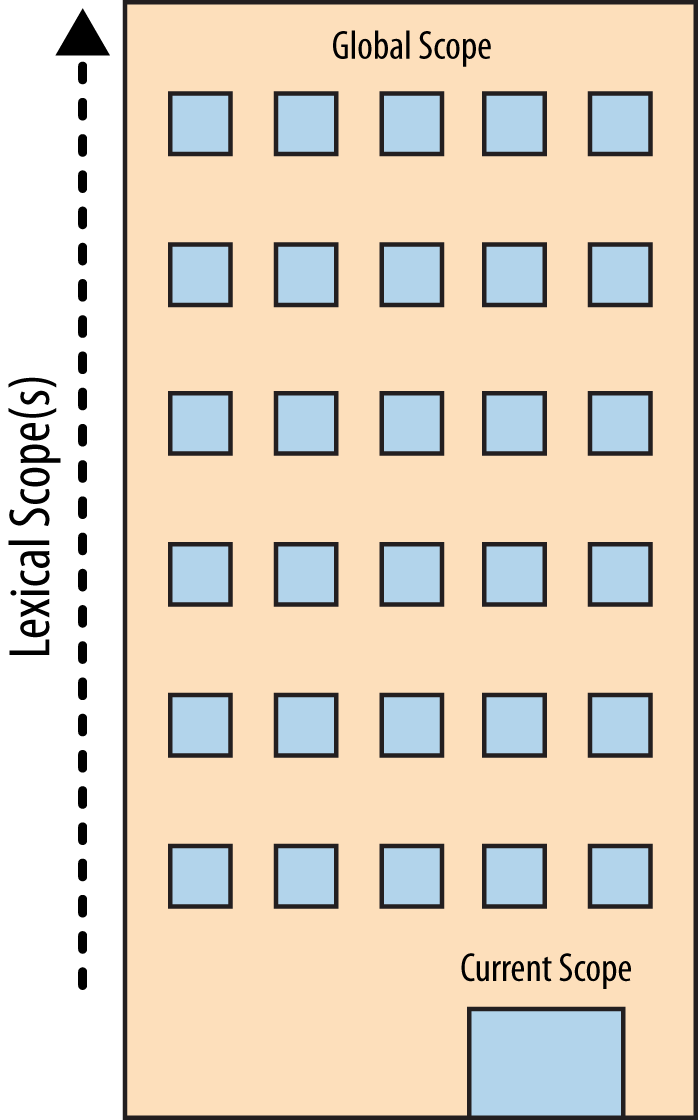
***Engine***: "Hey, Scope outside of foo, oh you're the global Scope, ok cool. Ever heard of b? Got an RHS reference for it."

***Scope***: "Yep, sure have. Here ya go."

The simple rules for traversing nested Scope: Engine starts at the currently executing Scope, looks for the variable there, then if not found, keeps going up one level, and so on. If the outermost global scope is reached, the search stops, whether it finds the variable or not.

### Building on Metaphors

To visualize the process of nested Scope resolution, I want you to think of this tall building.

[](https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/fig1.png)

The building represents our program's nested Scope rule set. The first floor of the building represents your currently executing Scope, wherever you are. The top level of the building is the global Scope.

You resolve LHS and RHS references by looking on your current floor, and if you don't find it, taking the elevator to the next floor, looking there, then the next, and so on. Once you get to the top floor (the global Scope), you either find what you're looking for, or you don't. But you have to stop regardless.

## Errors

Why does it matter whether we call it LHS or RHS?

Because these two types of look-ups behave differently in the circumstance where the variable has not yet been declared (is not found in any consulted Scope).

Consider:

function foo(a) {

console.log( a + b );

b = a;

}

foo( 2 );

When the RHS look-up occurs for b the first time, it will not be found. This is said to be an "undeclared" variable, because it is not found in the scope.

If an RHS look-up fails to ever find a variable, anywhere in the nested Scopes, this results in a ReferenceError being thrown by the Engine. It's important to note that the error is of the type ReferenceError.

By contrast, if the Engine is performing an LHS look-up and arrives at the top floor (global Scope) without finding it, and if the program is not running in "Strict Mode" [^note-strictmode], then the global Scope will create a new variable of that name **in the global scope**, and hand it back to Engine.

"No, there wasn't one before, but I was helpful and created one for you."

"Strict Mode" [^note-strictmode], which was added in ES5, has a number of different behaviors from normal/relaxed/lazy mode. One such behavior is that it disallows the automatic/implicit global variable creation. In that case, there would be no global Scope'd variable to hand back from an LHS look-up, and Engine would throw a ReferenceError similarly to the RHS case.

Now, if a variable is found for an RHS look-up, but you try to do something with its value that is impossible, such as trying to execute-as-function a non-function value, or reference a property on a null or undefined value, then Engine throws a different kind of error, called a TypeError.

ReferenceError is Scope resolution-failure related, whereas TypeError implies that Scope resolution was successful, but that there was an illegal/impossible action attempted against the result.

## Review

Scope is the set of rules that determines where and how a variable (identifier) can be looked-up. This look-up may be for the purposes of assigning to the variable, which is an LHS (left-hand-side) reference, or it may be for the purposes of retrieving its value, which is an RHS (right-hand-side) reference.

LHS references result from assignment operations. Scope-related assignments can occur either with the = operator or by passing arguments to (assign to) function parameters.

The JavaScript Engine first compiles code before it executes, and in so doing, it splits up statements like var a = 2; into two separate steps:

1. First, var a to declare it in that Scope. This is performed at the beginning, before code execution.
2. Later, a = 2 to look up the variable (LHS reference) and assign to it if found.

Both LHS and RHS reference look-ups start at the currently executing Scope, and if need be (that is, they don't find what they're looking for there), they work their way up the nested Scope, one scope (floor) at a time, looking for the identifier, until they get to the global (top floor) and stop, and either find it, or don't.

Unfulfilled RHS references result in ReferenceErrors being thrown. Unfulfilled LHS references result in an automatic, implicitly-created global of that name (if not in "Strict Mode" [^note-strictmode]), or a ReferenceError (if in "Strict Mode" [^note-strictmode]).

### Quiz Answers

function foo(a) {

var b = a;

return a + b;

}

var c = foo( 2 );

1. Identify all the LHS look-ups (there are 3!).

**c = .., a = 2 (implicit param assignment) and b = ..**

1. Identify all the RHS look-ups (there are 4!).

**foo(2.., = a;, a + .. and .. + b**

# Chapter 2: Lexical Scope

In Chapter 1, we defined "scope" as the set of rules that govern how the Engine can look up a variable by its identifier name and find it, either in the current Scope, or in any of the Nested Scopes it's contained within.

There are two predominant models for how scope works. The first of these is by far the most common, used by the vast majority of programming languages. It's called **Lexical Scope**, and we will examine it in-depth. The other model, which is still used by some languages (such as Bash scripting, some modes in Perl, etc.) is called **Dynamic Scope**.

Dynamic Scope is covered in Appendix A. I mention it here only to provide a contrast with Lexical Scope, which is the scope model that JavaScript employs.

## Lex-time

As we discussed in Chapter 1, the first traditional phase of a standard language compiler is called lexing (aka, tokenizing). If you recall, the lexing process examines a string of source code characters and assigns semantic meaning to the tokens as a result of some stateful parsing.

It is this concept which provides the foundation to understand what lexical scope is and where the name comes from.

To define it somewhat circularly, lexical scope is scope that is defined at lexing time. In other words, lexical scope is based on where variables and blocks of scope are authored, by you, at write time, and thus is (mostly) set in stone by the time the lexer processes your code.

**Note:** We will see in a little bit there are some ways to cheat lexical scope, thereby modifying it after the lexer has passed by, but these are frowned upon. It is considered best practice to treat lexical scope as, in fact, lexical-only, and thus entirely author-time in nature.

Let's consider this block of code:

function foo(a) {

var b = a \* 2;

function bar(c) {

console.log( a, b, c );

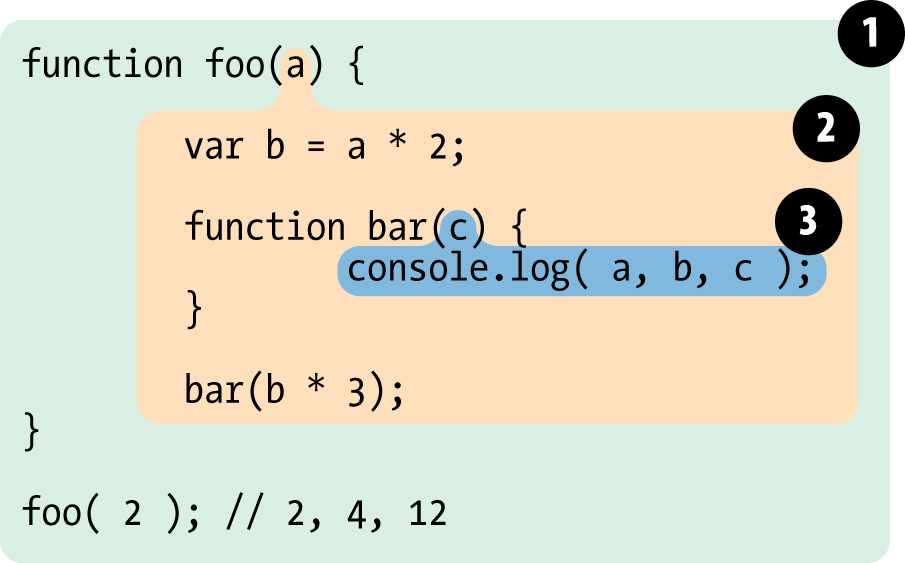
}

bar(b \* 3);

}

foo( 2 ); // 2 4 12

There are three nested scopes inherent in this code example. It may be helpful to think about these scopes as bubbles inside of each other.

[](https://github.com/getify/You-Dont-Know-JS/blob/master/scope%20&%20closures/fig2.png)

**Bubble 1** encompasses the global scope, and has just one identifier in it: foo.

**Bubble 2** encompasses the scope of foo, which includes the three identifiers: a, bar and b.

**Bubble 3** encompasses the scope of bar, and it includes just one identifier: c.

Scope bubbles are defined by where the blocks of scope are written, which one is nested inside the other, etc. In the next chapter, we'll discuss different units of scope, but for now, let's just assume that each function creates a new bubble of scope.

The bubble for bar is entirely contained within the bubble for foo, because (and only because) that's where we chose to define the function bar.

Notice that these nested bubbles are strictly nested. We're not talking about Venn diagrams where the bubbles can cross boundaries. In other words, no bubble for some function can simultaneously exist (partially) inside two other outer scope bubbles, just as no function can partially be inside each of two parent functions.

### Look-ups

The structure and relative placement of these scope bubbles fully explains to the Engine all the places it needs to look to find an identifier.

In the above code snippet, the Engine executes the console.log(..) statement and goes looking for the three referenced variables a, b, and c. It first starts with the innermost scope bubble, the scope of the bar(..) function. It won't find athere, so it goes up one level, out to the next nearest scope bubble, the scope of foo(..). It finds a there, and so it uses that a. Same thing for b. But c, it does find inside of bar(..).

Had there been a c both inside of bar(..) and inside of foo(..), the console.log(..) statement would have found and used the one in bar(..), never getting to the one in foo(..).

**Scope look-up stops once it finds the first match**. The same identifier name can be specified at multiple layers of nested scope, which is called "shadowing" (the inner identifier "shadows" the outer identifier). Regardless of shadowing, scope look-up always starts at the innermost scope being executed at the time, and works its way outward/upward until the first match, and stops.

**Note:** Global variables are also automatically properties of the global object (window in browsers, etc.), so it is possible to reference a global variable not directly by its lexical name, but instead indirectly as a property reference of the global object.

window.a

This technique gives access to a global variable which would otherwise be inaccessible due to it being shadowed. However, non-global shadowed variables cannot be accessed.

No matter where a function is invoked from, or even how it is invoked, its lexical scope is **only** defined by where the function was declared.

The lexical scope look-up process only applies to first-class identifiers, such as the a, b, and c. If you had a reference to foo.bar.baz in a piece of code, the lexical scope look-up would apply to finding the foo identifier, but once it locates that variable, object property-access rules take over to resolve the bar and baz properties, respectively.

## Cheating Lexical

If lexical scope is defined only by where a function is declared, which is entirely an author-time decision, how could there possibly be a way to "modify" (aka, cheat) lexical scope at run-time?

JavaScript has two such mechanisms. Both of them are equally frowned-upon in the wider community as bad practices to use in your code. But the typical arguments against them are often missing the most important point: **cheating lexical scope leads to poorer performance.**

Before I explain the performance issue, though, let's look at how these two mechanisms work.

### eval

The eval(..) function in JavaScript takes a string as an argument, and treats the contents of the string as if it had actually been authored code at that point in the program. In other words, you can programmatically generate code inside of your authored code, and run the generated code as if it had been there at author time.

Evaluating eval(..) (pun intended) in that light, it should be clear how eval(..) allows you to modify the lexical scope environment by cheating and pretending that author-time (aka, lexical) code was there all along.

On subsequent lines of code after an eval(..) has executed, the Engine will not "know" or "care" that the previous code in question was dynamically interpreted and thus modified the lexical scope environment. The Engine will simply perform its lexical scope look-ups as it always does.

Consider the following code:

function foo(str, a) {

eval( str ); // cheating!

console.log( a, b );

}

var b = 2;

foo( "var b = 3;", 1 ); // 1 3

The string "var b = 3;" is treated, at the point of the eval(..) call, as code that was there all along. Because that code happens to declare a new variable b, it modifies the existing lexical scope of foo(..). In fact, as mentioned above, this code actually creates variable b inside of foo(..) that shadows the b that was declared in the outer (global) scope.

When the console.log(..) call occurs, it finds both a and b in the scope of foo(..), and never finds the outer b. Thus, we print out "1 3" instead of "1 2" as would have normally been the case.

**Note:** In this example, for simplicity's sake, the string of "code" we pass in was a fixed literal. But it could easily have been programmatically created by adding characters together based on your program's logic. eval(..) is usually used to execute dynamically created code, as dynamically evaluating essentially static code from a string literal would provide no real benefit to just authoring the code directly.

By default, if a string of code that eval(..) executes contains one or more declarations (either variables or functions), this action modifies the existing lexical scope in which the eval(..) resides. Technically, eval(..) can be invoked "indirectly", through various tricks (beyond our discussion here), which causes it to instead execute in the context of the global scope, thus modifying it. But in either case, eval(..) can at runtime modify an author-time lexical scope.

**Note:** eval(..) when used in a strict-mode program operates in its own lexical scope, which means declarations made inside of the eval() do not actually modify the enclosing scope.

function foo(str) {

"use strict";

eval( str );

console.log( a ); // ReferenceError: a is not defined

}

foo( "var a = 2" );

There are other facilities in JavaScript which amount to a very similar effect to eval(..). setTimeout(..) and setInterval(..) can take a string for their respective first argument, the contents of which are evaluated as the code of a dynamically-generated function. This is old, legacy behavior and long-since deprecated. Don't do it!

The new Function(..) function constructor similarly takes a string of code in its **last** argument to turn into a dynamically-generated function (the first argument(s), if any, are the named parameters for the new function). This function-constructor syntax is slightly safer than eval(..), but it should still be avoided in your code.

The use-cases for dynamically generating code inside your program are incredibly rare, as the performance degradations are almost never worth the capability.

### with

The other frowned-upon (and now deprecated!) feature in JavaScript which cheats lexical scope is the with keyword. There are multiple valid ways that with can be explained, but I will choose here to explain it from the perspective of how it interacts with and affects lexical scope.

with is typically explained as a short-hand for making multiple property references against an object without repeating the object reference itself each time.

For example:

var obj = {

a: 1,

b: 2,

c: 3

};

// more "tedious" to repeat "obj"

obj.a = 2;

obj.b = 3;

obj.c = 4;

// "easier" short-hand

with (obj) {

a = 3;

b = 4;

c = 5;

}

However, there's much more going on here than just a convenient short-hand for object property access. Consider:

function foo(obj) {

with (obj) {

a = 2;

}

}

var o1 = {

a: 3

};

var o2 = {

b: 3

};

foo( o1 );

console.log( o1.a ); // 2

foo( o2 );

console.log( o2.a ); // undefined

console.log( a ); // 2 -- Oops, leaked global!

In this code example, two objects o1 and o2 are created. One has an a property, and the other does not. The foo(..)function takes an object reference obj as an argument, and calls with (obj) { .. } on the reference. Inside the withblock, we make what appears to be a normal lexical reference to a variable a, an LHS reference in fact (see Chapter 1), to assign to it the value of 2.

When we pass in o1, the a = 2 assignment finds the property o1.a and assigns it the value 2, as reflected in the subsequent console.log(o1.a) statement. However, when we pass in o2, since it does not have an a property, no such property is created, and o2.a remains undefined.

But then we note a peculiar side-effect, the fact that a global variable a was created by the a = 2 assignment. How can this be?

The with statement takes an object, one which has zero or more properties, and **treats that object as if**it**is a wholly separate lexical scope**, and thus the object's properties are treated as lexically defined identifiers in that "scope".

**Note:** Even though a with block treats an object like a lexical scope, a normal var declaration inside that with block will not be scoped to that with block, but instead the containing function scope.

While the eval(..) function can modify existing lexical scope if it takes a string of code with one or more declarations in it, the with statement actually creates a **whole new lexical scope** out of thin air, from the object you pass to it.

Understood in this way, the "scope" declared by the with statement when we passed in o1 was o1, and that "scope" had an "identifier" in it which corresponds to the o1.a property. But when we used o2 as the "scope", it had no such a"identifier" in it, and so the normal rules of LHS identifier look-up (see Chapter 1) occurred.

Neither the "scope" of o2, nor the scope of foo(..), nor the global scope even, has an a identifier to be found, so when a = 2 is executed, it results in the automatic-global being created (since we're in non-strict mode).

It is a strange sort of mind-bending thought to see with turning, at runtime, an object and its properties into a "scope" with"identifiers". But that is the clearest explanation I can give for the results we see.

**Note:** In addition to being a bad idea to use, both eval(..) and with are affected (restricted) by Strict Mode. with is outright disallowed, whereas various forms of indirect or unsafe eval(..) are disallowed while retaining the core functionality.

### Performance

Both eval(..) and with cheat the otherwise author-time defined lexical scope by modifying or creating new lexical scope at runtime.

So, what's the big deal, you ask? If they offer more sophisticated functionality and coding flexibility, aren't these goodfeatures? **No.**

The JavaScript Engine has a number of performance optimizations that it performs during the compilation phase. Some of these boil down to being able to essentially statically analyze the code as it lexes, and pre-determine where all the variable and function declarations are, so that it takes less effort to resolve identifiers during execution.

But if the Engine finds an eval(..) or with in the code, it essentially has to assume that all its awareness of identifier location may be invalid, because it cannot know at lexing time exactly what code you may pass to eval(..) to modify the lexical scope, or the contents of the object you may pass to with to create a new lexical scope to be consulted.

In other words, in the pessimistic sense, most of those optimizations it would make are pointless if eval(..) or with are present, so it simply doesn't perform the optimizations at all.

Your code will almost certainly tend to run slower simply by the fact that you include an eval(..) or with anywhere in the code. No matter how smart the Engine may be about trying to limit the side-effects of these pessimistic assumptions, **there's no getting around the fact that without the optimizations, code runs slower.**

## Review

Lexical scope means that scope is defined by author-time decisions of where functions are declared. The lexing phase of compilation is essentially able to know where and how all identifiers are declared, and thus predict how they will be looked-up during execution.

Two mechanisms in JavaScript can "cheat" lexical scope: eval(..) and with. The former can modify existing lexical scope (at runtime) by evaluating a string of "code" which has one or more declarations in it. The latter essentially creates a whole new lexical scope (again, at runtime) by treating an object reference as a "scope" and that object's properties as scoped identifiers.

The downside to these mechanisms is that it defeats the Engine's ability to perform compile-time optimizations regarding scope look-up, because the Engine has to assume pessimistically that such optimizations will be invalid. Code will run slower as a result of using either feature. **Don't use them.**

**Вы не знаете JS** под авторством Кайла Cимсона

**Предисловие**

Я уверен, вы заметили, что «JS» в заголовках серии этих книг не аббревиатура слов используемых для проклятия самого JavaScript, а для обозначения причуд этого языка.

С самого появления Интернета, JavaScript был основополагающей технологией, которая позволяла управлять интерактивностью вокруг контента, который мы используем. Начинался JavaScript с мерцающих траекторий движения компьютерной мышки и раздражающий всплывающих подсказок, но, почти через два десятилетия, технологии и возможности JavaScript выросли на несколько порядков, и сейчас мало кто усомниться в важности языка, стоящего в основе самой широкодоступной программной платформы в мире: Интернет.

Но, как язык, он постоянно был объектом сильной критики, частично из-за своего наследия, но даже больше из-за философии, стоящей в основе дизайна языка. Как сказал Брэндан Эйч: «Даже название этого языка вызывает ощущение статуса ребенка рядом со своим более зрелым братом Java.» Однако это просто случайность, возникшая из-за политики и маркетинга. Эти два языка во многом отличаются друг от друга. JavaScript имеет такое же отношение к Java, как «карнавал» к «автомобилю».

Поскольку JavaScript заимствует понятия и синтаксические идиомы с нескольких языков, в том числе сразу заметные процедурные корни языка С и менее очевидные функциональные корни языков Scheme или Lisp. JS доступен широкой аудитории разработчиков, даже тех у кого нет опыта программирования вовсе. Код выводящий на экран «Привет Мир» настолько прост, что язык выглядит очень притягательным и удобным при первом знакомстве.

Хоть JavaScript – это, пожалуй, один из самых простых языков, с которым можно быстро начать работать, но его эксцентричность делают твердое знание языка куда менее обычным явлением среди разработчиков, чем во многих других языках. Там, где требуется достаточно глубокое познание языка, такого как С или С++, для написания полномасштабной программы, полномасштабная разработка на JavaScript почти не требует полных знаний о всех возможностях языка.

Сложные концепции, которые глубоко укоренены в языке кажутся упрощениями, такими как передача функций в качестве обратных вызовов, что побуждает разработчика JavaScript просто использовать язык как есть и не слишком беспокоится о том, что происходит «под капотом».

Это одновременно привлекательный и простой в использовании язык, а также сложный и тонкий набор языковых механик, которые без тщательного изучения ускользнут от истинного понимания даже самых опытных разработчиков JavaScript.

В это и заключается парадокс JavaScript, его ахиллесова пята, которую мы имеем на сегодняшний день. Поскольку JavaScript можно использовать без его понимания, это самое понимание никогда не достигается.

**Цель**

Если при первом же моменте, когда вы сталкиваетесь с неожиданностью или беспомощностью в JavaScript, вашей реакцией будет добавить его в черный список, как многие привыкли, вы так и не постигните истинную сущность богатства JavaScript.

Хотя это подмножество языка было известно как «Хорошие части», я умоляю вас, дорогой читатель, рассмотреть его как «Простые части», «Безопасные части» или даже «Неполные части».

Эта серия книг «Вы не знаете JS» предлагает противоположную задачу: изучить и глубоко понять весь JavaScript, в особенности «Сложные части».

Здесь мы обращаемся к общей тенденции среди разработчиков JS: узнать «достаточно», чтобы работать, не заставляя себя точно узнать, как и почему язык ведет себя тем или иным образом и как это осуществляется. Кроме того, мы избегаем типичного совета отступить, если путь становится слишком сложным.

Я не доволен и не должен быть, останавливаясь на том, что что-то работает и не важно как. Я подталкиваю вас по этой ухабистой и непротоптанной дороге и изучить все возможности JavaScript. С этими знаниями, ни одна техника, ни один фреймворк, ни один акроним недели не озадачат вас.

В этих книгах берутся конкретные ключевые части языка, которые чаще всего неправильно понимаются или недооцениваются, и исчерпывающе изучаются. В должны заканчивать чтение с твердой уверенностью в своем понимании не только теории, но и необходимой практики.

JavaScript, известный вам сейчас, вероятно, является продуктом передачи вам знаний от других людей, с отсутствием полного понимания. Этот JS лишь тень реального языка. Вы еще не знаете JavaScript, но если вы углубитесь в эту серию книг, то эти знания появятся. Продолжайте читать, друзья. JavaScript ждет вас.

**Заключение**

JavaScript замечательный. Его легко изучить частично и гораздо труднее постичь полностью (или хотя бы в достаточной мере). Когда разработчики сталкиваются с путаницей, они обычно обвиняют язык, а не их непонимание. Эти книги направлены на то, чтобы исправить это, и высоко оценить язык, который вы можете и должны глубоко знать.

Примечание: Многие примеры в этой книге предполагают использование современных и будущих сред разработки JavaScript, такие как ES6. Некоторые вырезки кода могут работать не так, как описано, если они запускаются на старых (до ES6) движках.

# Глава 1: Что такое область видимости?

Одной из наиболее фундаментальных парадигм почти всех языков программирования является способность сохранять значения в переменных, а затем извлекать и изменять эти значения. Фактически, способность сохранять и извлекать значения переменных является тем, что описывает состояние программы.

Без такой концепции программа могла бы выполнять некоторые задачи, но они были бы крайне ограниченными и не очень интересными.

Но включение переменных в нашу программу порождает самые интересные вопросы, которые мы сейчас рассмотрим. Где живут переменные? Другими словами, где они хранятся. И самое важное, как наша программа находит их, когда они ей необходимы.

Эти вопросы говорят о необходимости четко определенного набора правил хранения переменных в некотором месте для поиска этих переменных в дальнейшем. Мы будем называть этот набор правил областью видимости.

Но где и как эти правила области видимости устанавливаются?

## Теория компилятора

Это может быть очевидным или наоборот удивительным, в зависимости от вашего опыта работы с другими языками программирования, но, несмотря на то, что JavaScript подпадает под общую категорию «динамических» или «интерпретируемых» языков, на самом деле это компилируемый язык программирования. Он не компилируется заранее, как многие традиционный компилируемые языки программирования, так же как и результаты компиляции, переносимые между различными распределенными системами.

Но, тем не менее, движок JavaScript выполняет многие из тех шагов, хотя и более сложными способами, чем любой другой компилируемый язык программирования.

В обычном процессе компилируемого языка часть вашего программного кода будет выполняться в три шага, прежде чем она будет выполнена, этот процесс грубо назван компиляцией:

1. **Токенизация/Лексировка:** разбиение строки символов на значимые (для языка) куски, называемые токенами. Например, рассмотрим программу: var a = 2;. Эта программа, вероятно, будет разбита на следующие токены: var, a, =, 2 и ;. Пробелы могут сохраняться или не сохраняться в виде токена, в зависимости от того, является ли они значимыми или нет.

Примечание: Разница между токенизацией и лексировкой очень тонкая и академическая, она заключается в том, как идентифицированы эти токены: с сохранением состояния или без него. Проще говоря, если токенизатор должен был вызвать правила синтаксического анализа с сохранением состояния, чтобы выяснить, следует ли считать его отдельным токеном или частью другого токена, который будет лексироваться.

1. **Анализ: принятие потока (массива) токенов и превращение его в дерево вложенных элементов, которые в совокупности представляют грамматическую структуру программы. Это дерево называется «АДС» (абстрактное дерево синтаксиса).**

Дерево для var a = 2; может начинаться с узла верхнего уровня, называемого Декларацией, с дочерним узлом под названием Идентификатор (значение которого равно а), а другой дочерний элемент – Присваивание, который в свою очередь имеет дочерний элемент Числовой Литерал (значение которого равно 2).

1. **Генерация кода:** процесс принятия АДС и превращение его в исполняемый код. Эта часть сильно варьируется в зависимости от языка, платформы, для которой этот код написан и т.д.

Таким образом, вместо того, чтобы погрязнуть в деталях, мы просто описываем способ как АДС для var a = 2; превратить в набор машинных инструкций, чтобы фактически создать переменную с именем а (включая резервную память), а затем сохранить в нее значение.

Примечание: Детали того, как движок управляет системными ресурсами находятся за пределами того, что мы будем изучать, поэтому мы считаем само собой разумеющимся, что движок может создавать и хранить переменные по мере необходимости.

Движок JavaScript намного сложнее, чем просто эти три шага, как и многие другие языковые компиляторы. Например, в процессе синтаксического анализа и генерации кода существуют определенные меры для оптимизации производительности выполнения, включая свертывание избыточных элементов и т.д.

Итак, я описал все в общих чертах. Но я думаю, что скоро вы увидите, почему эти знания, даже на таком высоком уровне, будут полезны.

Движки JavaScript не имеют такой роскоши как большое время для компиляции, в отличие от других языковых компиляторов, потому что компиляция JavaScript не выполняется предварительно на этапе сборки, как в других языках.

Для JavaScript компиляция происходит во многих случаях за микросекунды (или меньше) до того, как выполниться код. Чтобы обеспечить максимальную производительность JS-движки используют всевозможные трюки (например компиляция «на лету», лениво компилируется и даже имеет механизмы горячей рекомпиляции), которые находятся за гранью того, что здесь обсуждаем.

Скажем, для простоты, что любой фрагмент JavaScript должен быть скомпилирован раньше (как правило, непосредственно перед), того как он будет выполнен. Таким образом, JS-компилятор примет программу var a = 2; и сначала скомпилирует его, и лишь потом (как правило, сразу после компиляции) будет готов его выполнить.

## Понимание области видимости

То, как мы будем изучать область видимости как процесс разговора. Но кто будет разговаривать?

### В ролях:

Давайте познакомимся с персонажами, которые взаимодействую во время обработки программы var a = 2;, чтобы понять их разговор, который мы вскоре прослушаем:

1. **Движок:** отвечает за сборку и выполнение нашей JavaScript программы.
2. **Компилятор:** один из друзей движка, занимается всей грязной работой синтаксического анализа и генерации кода.
3. **Область видимости:** подруга Движка, она собирает и ведет список всех объявленных переменных и применяет строгий набор правил доступности идентификаторов относительно исполняемого в текущий момент кода.

Для того, чтобы полностью понять как работает JavaScript, вам необходимо подумать над тем, как думают, задают вопросы и отвечают на них Движок и его друзья.

### Вернемся назад:

Когда вы видите программу var a = 2;, вы скорее всего думаете об ней, как о едином выражении. Но это не то, как его видит наш новый друг Движок. Фактически, Движок видит два разных утверждения, первый из которых компилятор будет обрабатывать во время компиляции и второй, который Движок будет обрабатывать во время исполнения.

Итак, давайте разберемся, как Движок и его друзья будут обращаться к программе var a = 2;.

Первое, что Компилятор сделает с этой программой, - это выполнит лексирование, чтобы разбить ее на токены, которые затем будут собраны в дерево. Но когда компилятор переходит к генерации кода, он будет рассматривать эту программу несколько иначе, чем предполагалось.

Разумным предположением было бы то, что компилятор будет генерировать код, который можно было бы представить в виде следующего псевдокода: «Выделите память для переменной, назовите ее, а затем вставьте значение 2 в эту переменную». К сожалению, это не совсем точно.

На самом деле компилятор будет действовать следующим образом:

1. Встречая var a, компилятор запрашивает область видимости, чтобы увидеть, существует ли уже такая переменная для этой конкретной коллекции объектов. Если это так, компилятор игнорирует это объявление и двигается дальше. В противном случае компилятор запрашивает область видимости для объявления новой переменной, называемой, а для этой коллекции объектов.
2. Затем компилятор создает код для движка, чтобы в дальнейшем выполнить присвоение, а = 2. При запуске кода, движок сначала запрашивает у области видимости, есть ли в текущей коллекции объектов переменная названная а. Если это так, Движок использует эту переменную. В противном случае, Движок ищет ее в другом месте (см. Раздел «Вложенные области видимости» ниже).

Если Движок в конечном итоге находит переменную, он присваивает ей значение 2. Если нет, то он поднимает руку и кричит об ошибке.

Подытожим: во-первых, компилятор объявляет переменную (если она ранее не объявлена в текущей области видимости), во-вторых во время выполнения код он ищет переменную в области видимости и присваивает ей значение, если она была найдена.

### Речь Компилятора:

Нам нужно немного больше познакомиться с терминологией компилятора, перед тем как продолжать.

Когда Движок выполняет код, созданный Компилятором, он должен найти переменную а, чтобы убедиться, была ли она объявлена, и этот поиск связан с областью видимости. И результат этого поиска, исполняемого Движком, зависит от его типа.

В нашем случае говорится, что Движок будет выполнять поиск «ЛС» для переменной а. Другой тип поиска называется «ПС».

Бьюсь об заклад, вы можете догадаться, что значат «Л» и «П». Эти термины означают «Левая сторона» и «Правая сторона».

Сторона чего? Операции присваивания.

Другими словами, ЛС поиск выполняется когда переменная появляется в левой части операции присваивания, а ПС поиск когда в правой.

На самом деле, давайте немного уточним. ПС поиск, в нашей ситуации, неотличим от простого поиска значения некоторой переменной. В то время как ЛС поиск пытается найти сам контейнер переменных, чтобы назначить эту переменную. Таким образом, ПС вовсе не обозначает «правую сторону присвоения», это обозначает не ЛС.

Вы могли подумать, что ПС вместо этого обозначает «извлечь значение из…».

Давайте рассмотрим это подробнее.

Если я напишу:

console.log( a );

Ссылка на а задействует ПС, потому что здесь ничего не назначается. Вместо этого мы пытаемся найти значение а, чтобы передать его в console.log(…).

И наоборот:

a = 2;

Ссылка на а здесь будет использовать ЛС, потому что нам не важно, какое текущее значение переменной, мы просто хотим найти ее для операции присваивания = 2.

Примечание: ЛС и ПС значит «левая и правая сторона инструкции», но эта инструкции может быть не только операцией присваивания =. Существует несколько других типов инструкций, поэтому эти понятия лучше трактовать как «тот, кто является объектом инструкции (ЛС)» и «тот, кто является источником инструкции (ПС)».

Рассмотрим эту программу, в которой используется как ЛС, так и ПС:

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

Последняя строка, вызывающая foo(…), для исполнения этой функции требует использования ПС для поиска foo, это можно трактовать как «найдите значение foo и дайте мне его». Более того, (…) означает, что значение foo должно быть исполняемым, поэтому оно должно быть функцией.

Здесь есть скрытая, но очень важная инструкция. Вы заметили ее?

Возможно, вы не заметили добавленный а = 2 в этом фрагменте кода. Это происходит, когда значение 2 передается в качестве аргумента функции foo(…), и в этом случае значение 2 присваивается переменной а. Чтобы (неявно) назначить переменную а, выполняется ЛС поиск.

Здесь также есть ПС поиск значения переменной а, которое передается в console.log(…), требующий это значение. Это ПС поиск для объекта console, после которого ищется метод log у объекта console.

Наконец, мы можем концептуализировать то, что между ЛС и ПС существует обмен для передачи значения 2 (путем ПС поиска переменной а) в log(…). Мы может предположить, что внутри реализации метода log(…) есть параметры, первый из которых (возможно названный арг1) вызывает ЛС поиск перед присвоением значения 2 в него.

Примечание: Возможно, возникает желание концептуализировать объявление функции function foo(a) {…} как объявление и присвоение обычной переменной, например var foo; и foo = function(a) {…}. Таким образом будет создавать ощущение, что при таком объявлении функции будет использован ЛС поиск.

Однако, тонкая, но важная разница заключается в том, что Компилятор обрабатывает как декларацию так и присвоение во время генерации кода, так что когда Движок выполняет код, ему не нужно определять значение функции foo. Таким образом, думать об объявлении функции как о процедуре ЛС поиска в том виде, в каком мы это обсуждаем сейчас -абсолютно не уместно.

### Разговор Движка и Области видимости

function foo(a) {

console.log( a ); // 2

}

foo( 2 );

Давайте представим вышеприведенный обмен (который обрабатывает этот фрагмент кода) в виде разговора. Разговор будет выглядеть примерно так:

***Движок***: Эй, область видимости, у меня есть ПС ссылка на foo. Когда-нибудь слышала о нём?

***Область видимости***: Да, слышала.  Компилятор декларировал ее всего секунду назад. Это функция. Держи.

***Движок***: Отлично, спасибо! Хорошо, я исполняю foo.

***Движок***: Эй, область видимости, у меня есть ЛС ссылка на a, когда-нибудь слышала о ней?

***Область видимости***: Да, слышала. Компилятор декларировал ее как официальный параметр для foo совсем недавно. Держи.

***Движок***: Незаменима, как всегда, Область видимости. Спасибо снова. Теперь время назначить 2 для a.

***Движок***: Эй, Область видимости, извиняюсь за то, что отвлекаю тебя снова. Мне нужен ПС поиск класса console. Когда-нибудь слышала о нем?

***Область видимости***: Нет проблем, Движок, это то, чем я занимаюсь целыми днями. Да, у меня есть console. Он встроенный. Держи.

***Движок***: Отлично. Ищем log(..). Это функция.

***Движок***: Эй, Область видимости. Можешь мне помочь с ПС ссылкой на a. Мне кажется я помню о ней, Но хочу проверить еще раз.

***Область видимости***: Да, ты правд, Движок, это все также переменная без изменений. Держи.

***Движок***: Круто. Передаю значение a, равное 2, в log(..).

...

### Тест

Проверьте ваше понимание на настоящий момент. Обязательно сыграйте роль Движка и поучаствуйте в "беседе" с Областью видимости*:*

function foo(a) {

var b = a;

return a + b;

}

var c = foo( 2 );

1. Определите все ЛС поиски (их 3!).
2. Определите все ПС поиски (их 4!).

**Примечание:** См. обзор этой главы, чтобы узнать ответы на тест!

## Вложенная область видимости

Мы говорили, что Область видимости— это набор правил поиска переменных по их идентификатору. Однако, обычно бывает более одной Области видимости.

Также как блок или функция вкладывается внутрь другого блока или функции, области видимости вкладываются внутрь других областей. Поэтому, если переменную не найти в ближайшей области видимости, Движок заглядывает в следующую внешнюю по отношению к этой область видимости, продолжая так до тех пор, пока не найдет или пока не достигнет самой внешней (т.е. глобальной) области.

Пример:

function foo(a) {

console.log( a + b );

}

var b = 2;

foo( 2 ); // 4

ПС ссылка на b не может быть разрешена внутри функции foo, но она может быть разрешена в Области видимости, окружающей ее (в этом случае, глобальной).

Поэтому, еще раз пересмотрев беседы между Движком и Областью видимости, мы возможно услышим:

***Движок***: "Эй, Область видимости foo, что-нибудь слышала о b? У меня есть ПС ссылка на нее".

***Область видимости***: "Не-а, никогда не слышала о такой. Попробуй что-нибудь другое!"

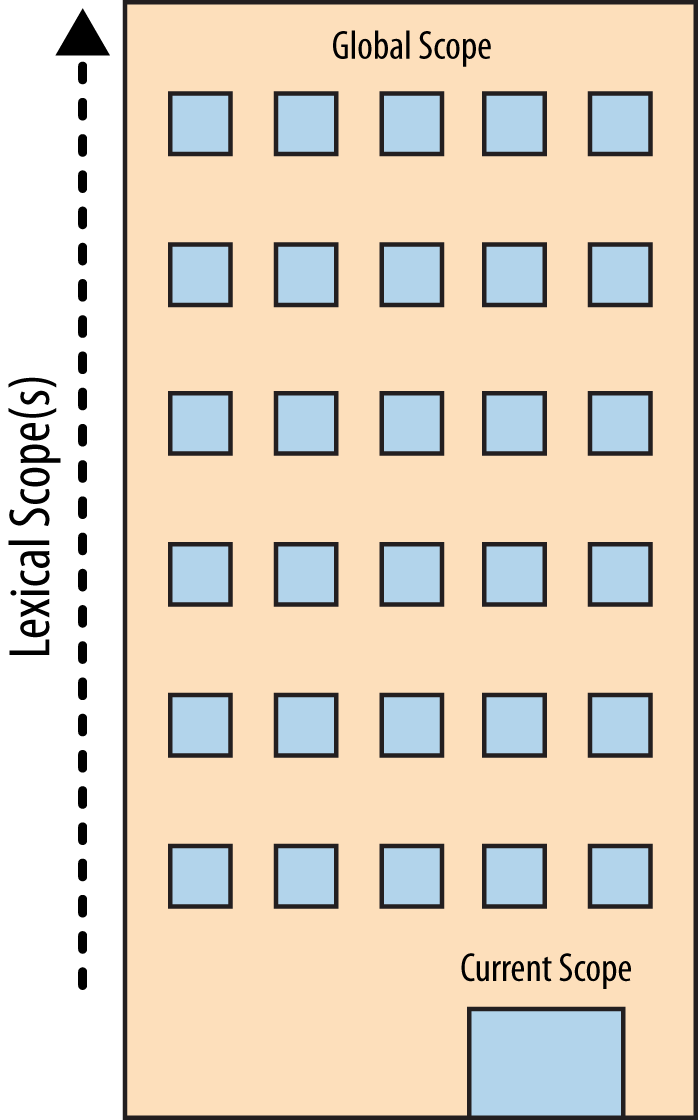
***Движок***: "Эй, Область видимости снаружи foo! О, ты еще и глобальная Область видимости, круто. Когда-нибудь слышала о b? У меня есть ПС ссылка на нее."

***Область видимости***: "Да-да, конечно есть. Держи."

Простые правила просмотра вложенных Областей видимости: Движок начинает в текущей выполняемой Области видимости, ищет в ней переменную, затем если не находит, продолжает поиск уровнем выше и так далее. Если достигнута самая внешняя глобальная область видимости, поиск останавливается, независимо от того, нашел он переменную или нет.

### Берем за основу метафоры

Для визуализации процесса разрешения во вложенных Областях видимости, я хочу, чтобы вы подумали об этом высоком здании.

[](https://github.com/azat-io/you-dont-know-js-ru/blob/master/scope%20&%20closures/fig1.png)

Здание символизирует набор правил вложенных Областей видимости нашей программы. Первый этаж здания представляет вашу текущую выполняемую Область видимости, где бы вы ни были. Верхний уровень здания — это глобальная Область видимости.

Вы разрешаете ЛС и ПС ссылки ища на вашем текущем этаже, а если вы не нашли что искали, поднимаетесь на лифте на следующий этаж, ища там, затем на следующий и так далее. Как только вы попадаете на верхний этаж (глобальная Область видимости), вы либо находите то, что искали, либо не находите. Но в любом случае вы должны остановиться.

## Ошибки

Почему имеет значение называть поиск ЛС или ПС?

Потому что эти два типа поиска ведут себя по-разному в обстановке, когда переменная еще не была объявлена (не была найдена ни в одной просмотренной Области видимости).

Представьте:

function foo(a) {

console.log( a + b );

b = a;

}

foo( 2 );

Когда происходит ПС поиск b первый раз, она не будет найдена. Это как бы "необъявленная" переменная, так как она не была найдена в этой области видимости.

Если RHS-поиск не сможет когда-либо найти переменную, в любой из вложенных Областей видимости, это приведет к возврату Движком ошибки ОшибкаСсылки. Важно отметить, что эта ошибка имеет тип ОшибкаСсылки.

Напротив, если Движок выполняет ЛС поиск и достигает верхнего этажа (глобальной Области видимости) и не находит ничего, и если программа не запущена в "строгом режиме", то затем глобальная Область видимости создаст новую переменную с таким именем **в глобальной области видимости** и передаст ее обратно Движку.

"Нет, до этого не было ни одной, и я любезно создала ее для тебя."

"Строгий режим", который был добавлен в ES5, имеет ряд разных отличий от обычного/нестрогого/ленивого режима. Одно такое отличие — это то, что он запрещает автоматическое/неявное создание глобальных переменных. В этом случае, не было бы никакой переменной в глобальной Области видимости, чтобы передать обратно от ЛС поиска, и Движок выбросит ОшибкаСсылки аналогично случаю с ПС.

Теперь, если переменная найдена в ходе ПС поиска, но вы пытаетесь сделать что-то с ее значением, что невозможно, например, пытаетесь выполнить как функцию не-функциональное значение или ссылаетесь на свойство значения нуль или неопределен, то Движок выдаст другой вид ошибки, называемый ОшибкаТипа.

ОшибкаСсылки — это сбой разрешения имени, связанный с Областью видимости, тогда как ОшибкаТипа подразумевает, что разрешение имени в Области видимости было успешным, но была попытка выполнения нелегального/невозможного действия с результатом.

## Обзор

Область видимости — это набор правил, которые определяют где и как переменная (идентификатор) могут быть найдены. Этот поиск может осуществляться для целей присваивания значения переменной, которая является ЛС ссылкой, или может осуществляться для целей извлечения ее значения, которое является ПС ссылкой.

ЛС ссылки являются результатом операции присваивания. Присваивания, связанные с Областью видимости, могут происходить либо с помощью операции =, либо передачей аргументов (присваиванием) параметрам функции.

JavaScript Движок перед выполнением сначала компилирует код, и пока он это делает, он разбивает операторы, подобные var a = 2; на два отдельных шага:

1. Первый, var a, чтобы объявить ее в Область видимости. Это выполняется в самом начале, до исполнения кода.
2. Позже, a = 2 ищет переменную (ЛС ссылку) и присваивает ей значение, если находит.

Оба поиска ссылок ЛС и ПС начинаются в текущей выполняющейся Области видимости и если нужно (т.е. они не нашли что искали в ней), они работают с их более высокими вложенными Областями видимости, с одной областью (этажом) за раз, ища идентификатор, пока не доберутся до глобальной (верхний этаж) и не остановятся, вне зависимости от результата поиска.

Невыполненные ПС ссылки приводят к выбросу ОшибкаСсылки. Невыполненные ЛС ссылки приводят к автоматической, неявно созданной переменной с таким именем (если не включен "Строгий режим"), либо к ОшибкаСсылки (если включен "Строгий режим").

### Ответы к тесту

function foo(a) {

var b = a;

return a + b;

}

var c = foo( 2 );

1. Определите все ЛС поиски (их 3!).

c = ..**,**a = 2**(неявное присваивание параметру) и**b = ..

1. Определите все RHS-поиски (их 4!).

**foo(2.., = a;, a + .. и .. + b**

# Глава 2: Лексическая область видимости

В главе 1, мы определили "область видимости" как набор правил, которые регулируют как Движок может искать переменную по ее имени идентификатора и найти ее либо в текущей Области видимости, либо в любой из вложенных Областей видимости, в которой она содержится.

Есть две преобладающих модели того, как работает область видимости. Первая из них, безусловно самая распространенная, используется необъятным большинством языков программирования. Она называется **Лексическая область видимости** и мы изучим ее в деталях. Другая модель, которая все еще используется некоторыми языками (такими как скриптовый Bash, некоторые режимы в Perl и т.д.), называется **Динамическая область видимости**.

Динамическая область видимости рассматривается в приложении A. Я упоминаю ее здесь только чтобы показать контраст с лексической областью действия, которая является моделью области видимости, используемой в JavaScript.

## Время разбора на лексемы

Как мы уже обсудили в главе 1, первая традиционная фаза работы стандартного компилятора языка называется разбиение на лексемы (лексировка или токенезация). Если вы припомните, то процесс разбиения на лексемы анализирует символы строки исходного кода и дает семантическое значение лексемам как результат некоторого разделения с состоянием.

Это и есть та концепция, которая предоставляет основу понимания что такое лексическая область видимости и откуда происходит ее название.

Определяя ее отчасти через саму себя, лексическая область видимости — это область видимости, которая определена во время разбора на лексемы. Иными словами, лексическая область видимости основана на том, где переменные и блоки области видимости были созданы вами во время написания и таким образом (в основном) навечно зафиксированы на момент, когда лексический анализатор обрабатывал ваш код.

**Примечание:** Совсем скоро мы увидим, что есть некоторые способы обмануть лексическую область действия, тем самым меняя ее после того, как лексический анализатор уже прошелся по ней, но к ним относятся неодобрительно. Считается лучшей практикой обращаться с лексической областью видимости как, по сути дела с чисто лексической и, следовательно, полностью относящейся к моменту написания кода по своей природе.

Давайте рассмотрим этот код:

function foo(a) {

var b = a \* 2;

function bar(c) {

console.log( a, b, c );

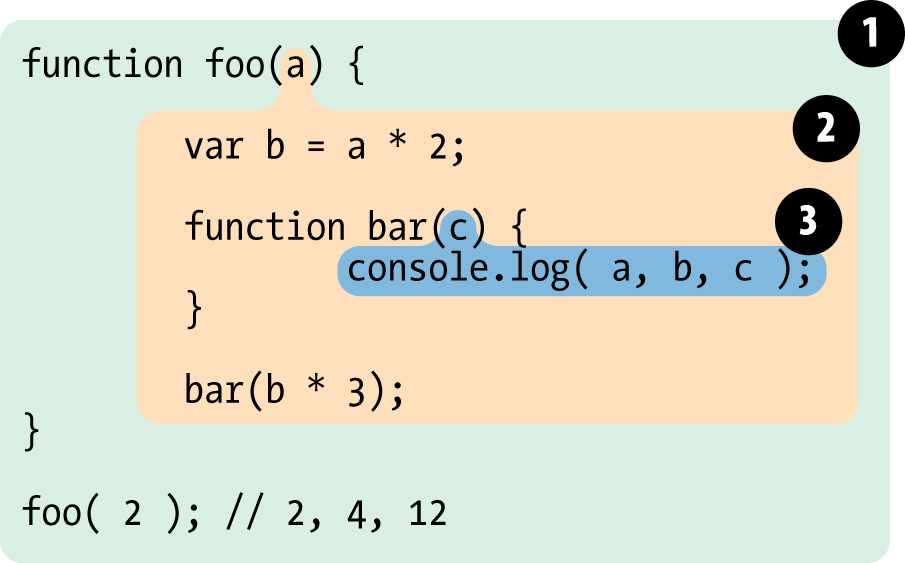
}

bar(b \* 3);

}

foo( 2 ); // 2 4 12

В этом примере кода есть три вложенных области видимости. Удобно представлять эти области видимости как зоны одна внутри другой.

[](https://github.com/azat-io/you-dont-know-js-ru/blob/master/scope%20&%20closures/fig2.png)

**Зона 1** заключает в себе глобальную область видимости, у которой есть всего один идентификатор: foo.

**Зона 2** заключает в себе область видимости foo, которая включает в себя три идентификатора: a, bar и b.

**Зона 3** заключает в себе область видимости bar и она включает в себя всего один идентификатор: c.

Зоны областей видимости определяются тем, где написаны блоки области видимости, которые последовательно вложены друг в друга. В следующей главе, мы обсудим различные единицы области видимости, а сейчас давайте просто допустим, что каждая функция создает новую зону области видимости.

Зона для bar полностью содержится в зоне для foo, потому что (и только поэтому) это то место, которое мы выбрали для создания функции bar.

Заметьте, что эти вложенные зоны вложены однозначно и четко. Мы не говорим сейчас о круговых диаграммах Эйлера—Венна, где зоны могут пересекать границы. Иными словами, ни одна зона для функции не может одновременно существовать (частично) внутри двух других зон внешних областей видимости, как и ни одна функция не может частично быть внутри каждой из двух родительских функциях.

### Поиски

Структура и относительное положение этих зон областей видимости полностью объясняет Движку все места, в которые ему нужно заглянуть, чтобы найти идентификатор.

В вышеприведенном коде, Движок выполняет оператор console.log(..) и идет искать три переменных a, b, и c, на которые есть ссылки. Сначала он начинает с самой внутренней зоны области видимости, области видимости функции bar(..). Он не найдет там a, поэтому пойдет на уровень выше, наружу к следующей ближайшей зоне области видимости, области видимости foo(..). Там он наконец найдет a, и поэтому использует эту a. То же самое и для b. А вот c он найдет внутри bar(..).

Если бы c была и внутри bar(..), и внутри foo(..), то оператор console.log(..) нашел и использовал ту, что в bar(..), никогда не трогая такую же из foo(..).

**Поиск в области видимости прекращается как только он находит первое совпадение**. Одно и то же имя идентификатора может быть указано в нескольких слоях вложенных областей видимости, что называется "затенение" (внутренний идентификатор "затеняет" внешний). Независимо от затенения, поиск в области видимости всегда начинается с самой внутренней области видимости, исполняющейся в данный момент и работает таким путем по направлению наружу/вверх пока не найдется первое совпадение и тогда останавливается.

**Примечание:** Глобальные переменные также автоматически являются свойствами глобального объекта (window в браузерах и т.п.), поэтому можно ссылаться на глобальную переменную не прямо по ее лексическому имени, а вместо этого косвенно использовать ссылку на свойство глобального объекта.

window.a

Эта техника дает доступ к глобальной переменной, которая в противном случае была бы недоступна из-за затенения. Однако, неглобальные затененные переменные не могут быть доступны.

Не важно откуда вызывается функция или даже как она вызывается, ее лексическая область видимости определена **только** тем, где функция была объявлена.

Процесс поиска в лексической области видимости применяется только к идентификаторам первого класса, таким как a, b и c. Если у вас есть ссылка на foo.bar.baz в строке кода, поиск в лексической области будет применен чтобы найти идентификатор foo, но как только он находит эту переменную, ему на смену приходят правила доступа к свойствам объекта, чтобы разрешить имена свойств bar и baz, соответственно.

## Обманываем лексическую область видимости

Если лексическая область видимости определяется только тем, где объявлена функция, что целиком во власти момента написания кода, какой может быть возможный путь "изменить" (т.е. обмануть) лексическую область во время выполнения?

В JavaScript ест два таких механизма. К ним обоим одинаково неодобрительно относятся в широком сообществе как к плохим практикам использования кода. Но типичные аргументы против них обычно не видят самого главного: **обман лексической области видимости ведет к более худшей производительности.**

Перед тем как я объясню проблему с производительностью, всё же, давайте взглянем на то, как работают эти два механизма.

### eval

Функция eval(..) в JavaScript берет строку как аргумент и интерпретирует содержимое строки как если бы это был код, написанный в этой точке программы. Другими словами, вы можете программно генерировать код внутри вашего собственного кода и запускать сгенерированный код как если бы он был там в время написания кода.

При вычислении eval(..) в таком свете, должно быть ясно как eval(..) позволяет модифицировать окружение лексической области видимости обманывая и притворяясь, что этот код был тут всё время.

На последующих строках кода после того, как выполнена eval(..), Движок не "узнает" или не "позаботится" о том, что предыдущий код, о котором идет речь, был динамически интерпретирован и таким образом изменил окружение лексической области видимости. Движок просто выполнит свои поиски по лексической области видимости как он обычно это делает.

Представьте следующий код:

function foo(str, a) {

eval( str ); // обман!

console.log( a, b );

}

var b = 2;

foo( "var b = 3;", 1 ); // 1, 3

Строка "var b = 3;" интерпретируется в точке вызова eval(..), как будто этот код был тут всегда. Поскольку этот код объявляет новую переменную b, он изменяет существующую лексическую область foo(..). Фактически, как было указано выше, этот код на самом деле создает переменную b внутри foo(..), которая затеняет b, которая была объявлена во внешней (глобальной) области видимости.

Когда происходит вызов console.log(..), он находит и a, иb в области видимости foo(..), но никогда не найдет внешнюю b. По этой причине, мы напечатаем "1, 3" вместо "1, 2" как это было бы в обычном случае.

**Примечание:** В этом примере для простоты строка "кода", которую мы передали, была фиксированным литералом. Но она легко может быть создана программно соединением символов вместе на основе логики вашей программы. eval(..) обычно используется для динамически созданного кода, поскольку динамическое вычисление по существу статического кода из строкового литерала не дает никакого реального преимущества перед простым написанием этого кода напрямую.

По умолчанию, если строка кода, которую выполняет eval(..), содержит одно или более объявлений (переменных или функций), тогда это действие меняет существующую лексическую область видимости, в которой располагается eval(..). Технически, eval(..) может быть вызвана "неявно", путем различных трюков (вне нашего обсуждения здесь), которые приводят к тому, что она вместо этого запускается в контексте глобальной области видимости, таким образом меняя ее. Но в любом случае, eval(..) может в время исполнения менять лексическую область видимости, определенную на момент написания кода.

**Примечание:** Когда eval(..) используется в программе, работающей в строгом режиме, она работает со своей собственной лексической областью видимости, что означает, что объявления, сделанные внутри eval(), не поменяют окружающую область видимости.

function foo(str) {

"use strict";

eval( str );

console.log( a ); // ОшибкаСсылки: a не определена

}

foo( "var a = 2" );

Есть другие средства в JavaScript, которые почти равносильны по эффекту вызовам eval(..). setTimeout(..) и setInterval(..), которые могут принимать строку как свой первый аргумент, содержимое которой вычисляется как код динамически сгенерированной функции. Это старая, унаследованная функциональность и давным-давно устаревшая. Не делайте так!

Конструктор функции new Function(..) аналогично принимает строку кода в своем **последнем** аргументе, чтобы превратить ее в динамически сгенерированную функцию (первые аргументы, если указаны, являются именованными параметрами для новой функции). Такой синтаксис конструктора функции немного безопаснее, чем eval(..), но его также следует избегать в вашем коде.

Варианты использования динамически сгенерированного кода внутри вашей программы крайне редки, поскольку снижение производительности почти никогда не стоит такой возможности.

### with

Еще одна возможность в JavaScript, к которой неодобрительно относятся (и которая сейчас устарела!), которая обманывает лексическую область видимости, это ключевое слово with. Есть много подходящих путей, чтобы объяснить что такое with, но я выберу объяснение с точки зрения того, как оно взаимодействует и влияет на лексическую область видимости.

with обычно описывают как сокращение для выполнения множественных ссылок на свойства объекта без повторения каждый раз ссылки на сам объект.

Например:

var obj = {

a: 1,

b: 2,

c: 3

};

// более "скучно" повторять "obj"

obj.a = 2;

obj.b = 3;

obj.c = 4;

// "легкое" сокращение

with (obj) {

a = 3;

b = 4;

c = 5;

}

Однако, здесь происходит нечто большее, чем просто удобное сокращение для доступа к свойствам объекта. Представьте:

function foo(obj) {

with (obj) {

a = 2;

}

}

var o1 = {

a: 3

};

var o2 = {

b: 3

};

foo( o1 );

console.log( o1.a ); // 2

foo( o2 );

console.log( o2.a ); // неопределен

console.log( a ); // 2 — Упс, утекшая глобальная переменная!

В этом примере кода, создаются два объекта o1 и o2. У одного есть свойство a, а у другого — нет. Функция foo(..)берет ссылку на объект obj как аргумент, а затем вызывает with (obj) { .. } с этой ссылкой. Внутри блока with мы делаем, как нам представляется, обычную лексическую ссылку на a, на самом деле LHS-ссылку (см. главу 1), чтобы присвоить ей значение 2.

Когда мы передаем o1, присвоение a = 2 находит свойство o1.a и присваивает ему значение 2, что нашло свое отражение в последующем операторе console.log(o1.a). Однако, когда мы передаем o2, поскольку у него нет свойства a, это свойство не создается, а o2.a остается неопределен.

Но затем мы замечаем своеобразный побочный эффект, факт того, что присвоение a = 2 создало глобальную переменную a. Как такое может быть?

Оператор with берет объект, у которого есть ноль или более свойств, и **трактует этот объект как если бы**он**являлся целиком отдельной лексической областью видимости**, и таким образом свойства объекта воспринимаются как лексически определенные идентификаторы в этой "области видимости".

**Примечание:** Даже если блок with трактует объект как лексическую область видимости, обычное объявление var внутри этого блока with не будет входить в область этого блока with, а вместо этого будет в области видимости функции, содержащей этот блок.

В то время как функция eval(..) может менять существующую лексическую область видимости, если она принимает строку кода с одним или более объявлениями в ней, то оператор with на самом деле создает **полностью новую лексическую область действия** на ровном месте из объекта, который вы ему передаете.

Таким образом мы поняли, что "область видимости", объявленная оператором with когда мы передали o1, была o1 и что у этой "области видимости" есть "идентификатор", который соответствует свойству o1.a. Но когда мы использовали o2 как "область видимости", у нее не было такого "идентификатора" a и поэтому сработали обычные правила поиска ЛС идентификатора (см. главу 1).

Ни "область видимости" o2, ни область видимости foo(..), ни даже глобальная область видимости не нашли у себя идентификатор a, поэтому, когда выполняется a = 2, это приводит к созданию автоматической глобальной переменной (поскольку мы в нестрогом режиме).

Это незнакомая и отчасти ошеломляющая мысль увидеть, как with превращает, во время выполнения, объект и его свойства в "область видимости" с "идентификаторами". Но это — самое понятное объяснение, которое я могу дать результатам, которые мы видели.

**Примечание:** В дополнение к тому, что является плохой идеей их использовать, как eval(..), так и withподвергаются воздействию (ограничиваются) строгим режимом. with полностью запрещено, в то время как различные формы скрытых или небезопасных eval(..) запрещены при сохранении базовой функциональности.

### Быстродействие

Оба eval(..) и with обманывают в той или иной форме лексическую область видимости, определенную на этапе кодирования, изменением или созданием новой лексической области видимости во время выполнения.

Итак, что тут такого, спросите вы? Если они предлагают улучшенную функциональность и гибкость кодирования, разве это не хорошие возможности? **Нет.**

В Движке JavaScript есть много оптимизаций быстродействия, которые он выполняет во время фазы компиляции. Некоторые из этих оптимизаций сводятся к возможности по сути статически анализировать код, как только он разбирается на лексемы, и заранее определять где находятся все переменные и функции, для того чтобы понадобилось меньше усилий для разрешения имен идентификаторов во время выполнения.

Но если Движок находит в коде eval(..) или with, он фактически должен предположить, что все его знание о местонахождении идентификаторов может быть неправильным, потому что он не знает точно во время написания кода какой код вы можете передать в eval(..), чтобы поменять лексическую область видимости или какое содержимое объекта вы можете передать в with, чтобы создать новую лексическую область видимости, чтобы быть в курсе.

Иными словами, с точки зрения пессимистического здравого смысла, большинство таких оптимизаций, которые он мог бы сделать, бессмысленны, если есть eval(..) или with, поэтому он просто не выполняет никаких оптимизаций.

Ваш код почти определенно будет стремиться работать медленнее просто из-за того факта, что вы включили eval(..)или with где-либо в вашем коде. Не важно насколько умным может быть Движок пытаясь ограничить побочные эффекты этих пессимистических предположений, но**нельзя обойти тот факт, что без оптимизаций код работает медленнее.**

## Обзор

Лексическая область видимости означает, что область видимости определена решениями о том, где объявляются функции на стадии написания кода. Фаза разбиения на лексемы при компиляции фактически способна узнать где и как объявлены все идентификаторы, и таким образом предсказать как их будут искать во время выполнения.

Два механизма в JavaScript могут "обмануть" лексическую область видимости: eval(..) и with. Первый может менять существующую лексическую область видимости (во время выполнения) исполняя строку "кода", в которой есть одно или несколько объявлений. Второй по сути создает целую новую лексическую область видимости (снова во время выполнения) интерпретируя ссылку на объект как "область видимости", а свойства этого объекта как идентификаторы этой области.

Недостаток этих механизмов в том, что они лишают смысла возможность Движка выполнять оптимизации во время компиляции, принимающие во внимание поиск в области видимости, так как Движок должен пессимистически предположить, что такие оптимизации будут неправильными. Код будет выполняться медленнее в результате использования любой из этих возможностей. **Не используйте их!**