

O'REILLY®

«Кайл критически переосмысливает каждую мелочь  
в языке, и его подход постепенно меняет  
ваш образ мышления и рабочий процесс».

— Шейн Хадсон, разработчик веб-сайтов

КАЙЛ СИМПСОН

# ЗАМЫКАНИЯ & ОБЪЕКТЫ

ВЫ НЕ ЗНАЕТЕ  
JS



ББК 32.988-02-018  
УДК 004.738.5  
С37

## **Симпсон К.**

С37 {Вы не знаете JS} Замыкания и объекты. — СПб.: Питер, 2019. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1255-5

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное, но при этом глубоко продуманное руководство познакомит вас с областями видимости, замыканиями, ключевым словом `this` и объектами — концепциями, которые необходимо знать для более эффективного и производительного программирования на JS. Вы узнаете, почему они работают и как замыкания могут стать эффективной частью вашего инструментария разработки.

Как и в других книгах серии «Вы не знаете JS», здесь показаны нетривиальные аспекты языка, от которых программисты JavaScript предпочитают держаться подальше. Вооружившись этими знаниями, вы достигнете истинного мастерства JavaScript.

**16+** (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988-02-018  
УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491904152 англ.

Authorized Russian translation of the English edition of You Don't Know JS: this & Object Prototypes (ISBN 9781491904152)  
© 2014 Getify Solutions, Inc.  
Authorized Russian translation of the English edition of You Don't Know JS: Scope & Closures (ISBN 9781449335588)  
© 2014 Getify Solutions, Inc.  
This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-4461-1255-5

© Перевод на русский язык ООО Издательство «Питер», 2019  
© Издание на русском языке, оформление  
ООО Издательство «Питер», 2019  
© Серия «Бестселлеры O'Reilly», 2019

# Оглавление

<b>Введение</b> .....	<b>11</b>
Задача .....	12
Благодарности .....	14
О книге .....	22
Типографские соглашения .....	23
Использование программного кода примеров .....	24
От издательства .....	24
 <b>ЧАСТЬ 1. ОБЛАСТЬ ВИДИМОСТИ И ЗАМЫКАНИЯ.....</b>	<b>25</b>
<b>Предисловие</b> .....	<b>26</b>
 <b>Глава 1. Что такое область видимости?.....</b>	<b>28</b>
Немного теории компиляторов .....	29
Разбираемся в областях видимости .....	31
Участники .....	31
Туда и обратно .....	32
Немного терминологии .....	33
Общение Движка с Областью видимости .....	36
Упражнение .....	37
Вложенная область видимости .....	38
Метафоры .....	39
Ошибки .....	41

Итоги. . . . .	.42
Ответ на упражнение. . . . .	.43
<b>Глава 2. Лексическая область видимости . . . . .</b>	<b>44</b>
Стадия лексического анализа. . . . .	.45
Поиск. . . . .	.47
Искажение лексической области видимости . . . . .	.48
eval. . . . .	.49
with . . . . .	.52
Быстродействие. . . . .	.55
Итоги. . . . .	.56
<b>Глава 3. Функциональные и блочные области видимости . . . . .</b>	<b>57</b>
Области видимости из функций . . . . .	.57
Как скрыться у всех на виду. . . . .	.59
Предотвращение конфликтов. . . . .	.61
Функции как области видимости. . . . .	.64
Анонимные и именованные функциональные выражения . . . . .	.66
Немедленный вызов функциональных выражений . . . . .	.67
Блоки как области видимости. . . . .	.70
with . . . . .	.72
try/catch . . . . .	.73
let. . . . .	.74
const. . . . .	.80
Итоги. . . . .	.80
<b>Глава 4. Поднятие. . . . .</b>	<b>82</b>
Курица или яйцо? . . . . .	.82
Компилятор наносит ответный удар. . . . .	.84
Сначала функции. . . . .	.87
Итоги. . . . .	.89

<b>Глава 5. Замыкание области видимости</b>	<b>90</b>
Просветление	91
Технические подробности	92
Теперь я вижу	96
Циклы и замыкания	99
Снова о блочной области видимости	102
Модули	103
Современные модули	109
Будущие модули	111
Итоги	113
<b>Приложение А. Динамическая область видимости</b>	<b>115</b>
<b>Приложение Б. Полифилы для блочной области видимости</b>	<b>118</b>
Traceur	120
Неявные и явные блоки	120
Быстродействие	123
<b>Приложение В. Лексическое this</b>	<b>124</b>
<b>ЧАСТЬ 2. THIS И ПРОТОТИПЫ ОБЪЕКТОВ</b>	<b>129</b>
<b>Предисловие</b>	<b>130</b>
<b>Глава 6. Что такое this?</b>	<b>133</b>
Для чего нужно this?	133
Путаница	135
Сама функция	136
Область видимости	141
Что такое this?	143
Итоги	144

**Глава 7. this обретает смысл! . . . . . 145**

Место вызова . . . . .	145
Ничего кроме правил . . . . .	147
Связывание по умолчанию . . . . .	147
Неявное связывание . . . . .	149
Явное связывание . . . . .	154
Связывание new . . . . .	158
Все по порядку . . . . .	161
Определение this . . . . .	166
Исключения связывания . . . . .	167
Игнорирование this . . . . .	167
Косвенные ссылки . . . . .	170
Мягкое связывание . . . . .	171
Лексическое поведение this . . . . .	173
Итоги . . . . .	176

**Глава 8. Объекты . . . . . 177**

Синтаксис . . . . .	177
Тип . . . . .	178
Встроенные объекты . . . . .	179
Содержимое . . . . .	182
Вычисление имен свойств . . . . .	184
Свойства и методы . . . . .	185
Массивы . . . . .	188
Дублирование объектов . . . . .	189
Дескрипторы свойств . . . . .	192
Неизменяемость . . . . .	197
[[Get]] . . . . .	200
[[Put]]. . . . .	202
Геттеры и сеттеры . . . . .	203
Существование . . . . .	206

Перебор. . . . .	209
Итоги. . . . .	215
<b>Глава 9. Классы. . . . .</b>	<b>217</b>
Теория классов . . . . .	218
Паттерн проектирования «класс» . . . . .	220
«Классы» JavaScript . . . . .	221
Механика классов . . . . .	222
Строительство . . . . .	222
Конструктор . . . . .	224
Наследование . . . . .	225
Полиморфизм . . . . .	228
Множественное наследование. . . . .	231
Примеси . . . . .	232
Явные примеси . . . . .	233
Неявные примеси . . . . .	241
Итоги. . . . .	242
<b>Глава 10. Прототипы . . . . .</b>	<b>244</b>
[[Prototype]]. . . . .	244
Object.prototype . . . . .	247
Назначение и замещение свойств . . . . .	247
«Класс» . . . . .	251
Функции «классов» . . . . .	251
«Конструкторы» . . . . .	256
Механика . . . . .	259
Наследование (на основе прототипов) . . . . .	263
Анализ связей «классов» . . . . .	268
Связи между объектами . . . . .	273
Создание связей вызовом Create(). . . . .	273
Связи как резерв? . . . . .	277
Итоги. . . . .	279

<b>Глава 11. Делегирование поведения</b>	<b>281</b>
Проектирование, ориентированное на делегирование	282
Теория классов	283
Теория делегирования	285
Сравнение моделей мышления	292
Классы и объекты	298
«Классы» виджетов	298
Делегирование для объектов Widget	302
Упрощение архитектуры	305
Расставание с классами	309
Более приятный синтаксис	312
Нелексичность	314
Интроспекция	316
Итоги	321
<b>Приложение Г. Классы ES6</b>	<b>322</b>
class	323
Проблемы class	325
Статический > динамический?	331
Итоги	332
<b>Об авторе</b>	<b>333</b>



# 1 Что такое область видимости?

Одна из самых фундаментальных парадигм почти всех языков программирования — возможность хранения значений в переменных и последующего чтения или изменения этих значений. Возможность сохранения и чтения значений из переменных — то, что образует *состояние* программы.

Без этой концепции программа сможет выполнять некоторые операции, но они будут в высшей степени ограниченными и не особенно интересными.

Однако включение переменных в программу поднимает самый интересный вопрос, которым мы сейчас займемся: где размещаются эти переменные? Другими словами, где они хранятся? И самое важное, как ваша программа находит их, когда в них возникнет необходимость?

Эти вопросы показывают, почему так необходим четко определенный набор правил для хранения переменных в определенном месте и их нахождения в будущем. Этот набор правил называется *областью видимости* (scope).

Но где и как задаются правила области видимости?

## Немного теории компиляторов

Это может быть очевидно, а может быть и удивительно, в зависимости от вашего опыта работы с разными языками, но несмотря на тот факт, что JavaScript относится к общей категории «динамических» или «интерпретируемых» языков, на самом деле это компилируемый язык. Код не компилируется заранее, как во многих традиционных компилируемых языках, а результаты компиляции не портируются между разными распределенными системами. Тем не менее движок JavaScript выполняет многие те же действия, что и любой традиционный компилятор (хотя и на более сложном уровне).

В традиционном процессе компиляции блок исходного кода — ваша программа — *перед* выполнением обычно проходит через три фазы обработки, которые приблизительно объединяются термином «компиляция»:

- *Лексический анализ/Разбиение на токены (Tokenizing/Lexing)* — разбиение последовательности символов на осмысленные (с точки зрения языка) фрагменты, называемые *токенами*. Для примера возьмем программу `var a = 2;`. Скорее всего, эта программа будет разбита на следующие токены: `var`, `a`, `=`, `2` и `;`. Пропуски могут сохраняться в виде токенов, а могут и не сохраняться в зависимости от того, имеет это смысл или нет.



Разница между *tokenizing* и *lexing* — вопрос достаточно тонкий и теоретический. Важно то, будет ли происходить идентификация токеном с состоянием или без. Проще говоря, если при вызове токенизатора активизируются правила разбора с состоянием, определяющие, должен ли данный токен считаться отдельным токеном или частью другого токена, это будет называться *lexing*.

- *Разбор (parsing)* — преобразование потока (массива) токенов в дерево вложенных элементов, которые в совокупности пред-

ставляют грамматическую структуру программы. Это дерево называется «абстрактным деревом синтаксиса», или AST (Abstract Syntax Tree). Скажем, дерево для `var a = 2;` может начинаться с узла верхнего уровня `VariableDeclaration`, который содержит дочерний узел `Identifier` (со значением `a`) и другой дочерний узел `AssignmentExpression`, у которого есть свой дочерний узел с именем `NumericLiteral` (его значение равно 2).

- *Генерирование кода* — процесс преобразования AST в исполняемый код. Эта часть сильно зависит от языка, целевой платформы и т. д.

Итак, вместо того чтобы вязнуть в подробностях, я просто скажу, что описанное ранее AST-дерево для `var a = 2;` может быть преобразовано в набор машинных команд для *создания* переменной с именем `a` (включая резервирование памяти и т. д.) и последующего сохранения значения в `a`.



Подробности того, как движок управляет системными ресурсами, выходят за рамки нашей темы. Поэтому будем просто считать, что движок способен создавать и сохранять переменные по мере необходимости.

Конечно, движок JavaScript не ограничивается *только* этими тремя этапами (как и большинство других компиляторов). Например, в процессе разбора и генерирования кода присутствуют фазы оптимизации кода, включая исключение избыточных элементов и т. д.

По этой причине я здесь даю общую картину. Но я думаю, вы вскоре увидите, почему все эти подробности (даже на высоком уровне) *важны* для нашего обсуждения.

В частности, движку JavaScript (в отличие от компиляторов других языков) недоступна такая роскошь, как достаточное время

для оптимизации, потому что компиляция JavaScript не выполняется в фазе построения заранее, как в других языках.

Для JavaScript компиляция во многих случаях выполняется за считанные микросекунды (и менее) до выполнения кода. Для обеспечения максимального быстродействия движка JS применяют всевозможные хитрости (например, JIT-компиляцию с отложенной компиляцией и даже оперативной перекомпиляцией и т. д.), которые выходят далеко за рамки «области видимости» нашего обсуждения.

Простоты ради будем считать, что любой фрагмент JavaScript должен компилироваться перед (обычно *непосредственно* перед) его выполнением. Итак, компилятор JS берет программу `var a = 2;`, *сначала* компилирует ее, а потом готовит ее к исполнению (обычно это происходит немедленно).

## Разбираемся в областях видимости

В своем изучении области видимости мы будем рассматривать процесс как некое подобие разговора. Но кто с кем ведет этот разговор?

### Участники

Сейчас мы познакомимся поближе с участниками, совместными усилиями обрабатывающими программу `var a = 2;`. Это поможет вам понять смысл их диалога, к которому мы вскоре прислушаемся:

- *Движок* — отвечает за всю компиляцию от начала до конца и выполнение программы JavaScript.
- *Компилятор* — один из друзей Движка; берет на себя всю черную работу по разбору и генерированию кода (см. предыдущий раздел).

- *Область видимости* — еще один друг Движка; собирает и ведет список всех объявленных идентификаторов (переменных) и устанавливает строгий набор правил их доступности для кода, выполняемого в данный момент.

Чтобы *в полной мере* понять, как работает JavaScript, вы должны начать думать как Движок (и его друзья), задавать себе те же вопросы, что и они, и давать на эти вопросы те же ответы.

## Туда и обратно

Когда вы видите программу `var a = 2;`, скорее всего, вы считаете, что она состоит из одной команды. Но с точки зрения Движка дело обстоит иначе. Движок видит здесь две разные команды: одну Компилятор обрабатывает во время компиляции, а другую Движок обрабатывает во время выполнения. Итак, разобьем на этапы процесс обработки программы `var a = 2;` Движком и другими компонентами.

Прежде всего, Компилятор проводит лексический анализ и разбирает программу на токены, которые затем разбираются в дерево. Но когда Компилятор добирается до генерирования кода, он рассматривает эту программу немного не так, как вы, возможно, ожидали.

Разумно было предположить, что Компилятор генерирует код, который можно было бы описать следующим псевдокодом: «Выделить память для переменной, присвоить ей метку `a` и сохранить в этой переменной значение `2`». К сожалению, такое описание не совсем точно.

Вместо этого Компилятор действует так:

1. Обнаруживая `var a`, Компилятор обращается к Области видимости, чтобы узнать, существует ли переменная `a` в наборе этой конкретной Области видимости. Если переменная существует,

то Компилятор игнорирует объявление и двигается дальше. В противном случае Компилятор обращается к Области видимости для объявления новой переменной с именем *a* в наборе этой области видимости.

2. Компилятор генерирует код для последующего выполнения Движком для обработки присваивания *a* = 2. Код, выполняемый Движком, сначала спрашивает у Области видимости, доступна ли переменная с именем *a* в наборе текущей области видимости. Если переменная доступна, то Движок использует эту переменную. Если нет, Движок ищет в *другом месте* (см. «Вложенная область видимости», с. 38).

Если Движок в конечном итоге находит переменную, он присваивает ей значение 2. Если нет, Движок поднимает тревогу и сообщает об ошибке.

Подведем итог: для присваивания значения переменной выполняются два разных действия. Сначала Компилятор объявляет переменную (если она не была объявлена ранее) в текущей Области видимости, а затем при выполнении Движок ищет переменную в Области видимости, и если переменная будет найдена — присваивает ей значение.

## Немного терминологии

Чтобы вы поняли суть происходящего далее, нам понадобятся некоторые специальные термины.

Когда Движок выполняет код, сгенерированный Компилятором на шаге 2, он должен провести поиск переменной *a* и определить, была ли она объявлена; этот поиск называется проверкой Области видимости. Однако тип проверки, выполняемой Движком, влияет на результат поиска.

В нашем примере Движок будет выполнять *LHS*-поиск переменной *a*. Другая разновидность поиска называется *RHS*. Сокращения означают «LeftHand Side» (левосторонний) и «RightHand Side» (правосторонний).

Левосторонний, правосторонний... по отношению к чему? К операции присваивания.

Иначе говоря, *LHS*-поиск выполняется при нахождении переменной в левой части операции присваивания, а *RHS*-поиск выполняется при нахождении переменной в правой части операции присваивания.

На самом деле стоит немного уточнить. Для наших целей *RHS*-поиск неотличим от простого поиска значения некоторой переменной, тогда как *LHS*-поиск пытается найти саму переменную-контейнер для присваивания. В этом отношении термин *RHS на самом деле* означает не «правую сторону присваивания» как таковую, а скорее «не левую сторону». В несколько упрощенном виде можно считать, что *RHS* означает «получить исходное значение».

А теперь разберемся подробнее.

В следующей команде:

```
console.log( a );
```

ссылка на *a* является *RHS*-ссылкой, потому что *a* здесь ничего не присваивается. Вместо этого мы собираемся прочитать значение *a*, чтобы значение могло быть передано `console.log(...)`.

Сравните:

```
a = 2;
```

Эта ссылка является *LHS*-ссылкой, потому что текущее значение переменной нас не интересует. Мы просто хотим найти переменную, которая могла бы послужить приемником для операции присваивания `= 2`.



«Левая/правая сторона присваивания» в обозначениях LHS и RHS не обязательно буквально означает «левая/правая сторона оператора присваивания =». Присваивание также может выполняться другими способами, поэтому лучше концептуально рассматривать их как «приемник присваивания» (LHS) и «источник присваивания» (RHS).

Возьмем следующую программу, в которой задействованы как LHS-, так и RHS-ссылки:

```
function foo(a) {  
    console.log( a ); // 2  
}  
  
foo( 2 );
```

Последняя строка с вызовом функции `foo(..)` также требует RHS-ссылки на `foo`, которая означает «Найти значение `foo` и предоставить его мне». Более того, `(..)` означает, что значение `foo` должно быть выполнено, а значит, это должна быть функция!

Здесь тоже выполняется неочевидное, но важное присваивание.

Возможно, вы упустили неявное присваивание `a = 2` в этом фрагменте кода. Оно происходит при передаче значения 2 в аргументе функции `foo(..)`, при котором значение 2 присваивается параметру `a`. Чтобы (неявно) присвоить значение параметру `a`, выполняется LHS-поиск.

Также здесь присутствует RHS-ссылка на значение `a`; полученное значение передается `console.log(..)`. Для выполнения `console.log(..)` тоже необходима ссылка. Сначала выполняется RHS-поиск объекта `console`, после чего поиск по свойствам определяет, существует ли среди них метод с именем `log`.

Наконец, можно на концептуальном уровне представить, что при передаче значения 2 (посредством RHS-поиска переменной `a`) методу `log(..)` происходит LHS/RHS-взаимодействие. Внутри



встроенной реализации `log(..)` можно считать, что у нее есть параметры, с первым из которых (вероятно, с именем `arg1`) выполняется LHS-поиск, перед тем как ему будет присвоено значение 2.



Возникает соблазн представить объявление функции `function foo(a) {...` как обычное объявление переменной с присваиванием, что-то вроде `var foo` и `foo = function(a){...}`. При таком представлении возникает столь же соблазнительная мысль считать, что при таком объявлении функции также действует LHS-поиск.

Однако здесь существует неочевидное, но важное различие: Компилятор обрабатывает объявление и определение значения во время генерирования кода, чтобы во время выполнения кода Движком «присваивание» значения функции `foo` не требовало никакой дополнительной обработки. А следовательно, на самом деле неправильно рассматривать объявление функции как присваивание с LHS-поиском в том смысле, в котором он здесь рассматривается.

## Общение Движка с Областью видимости

```
function foo(a) {  
    console.log( a ); // 2  
}  
foo( 2 );
```

Представим это взаимодействие (с обработкой этого фрагмента) в виде разговора. Общение между Движком и Областью видимости будет происходить примерно так:

*Движок:* Эй, Область видимости, у меня есть RHS-ссылка на `foo`. Знаешь, что это?

*Область видимости:* Ну да, знаю. Компилятор объявил `foo` секунду назад. Это функция. Вот, держи.

*Движок:* Здорово, отлично! Перехожу к выполнению `foo`.

*Движок:* Эй, Область видимости, у меня есть LHS-ссылка на `a`. Знаешь, что это?

*Область видимости:* Ну да, знаю. Компилятор только что объявил `a` как формальный параметр `foo`. Вот, держи.

*Движок:* Полезно, как всегда, Область видимости. Еще раз спасибо. А теперь пора присвоить `a` значение 2.

*Движок:* Эй, Область видимости, снова придется тебя побеспокоить. Мне нужно выполнить RHS-поиск для `console`. Знаешь, что это?

*Область видимости:* Без проблем, Движок, целыми днями только этим и занимаюсь. Да, я знаю, что такое `console` — это встроенный объект. Вот, держи.

*Движок:* Прекрасно. Теперь ищу `log( . . )`. Отлично, это функция.

*Движок:* Область видимости, а сможешь помочь с RHS-ссылкой на `a`? Вроде бы я помню что-то такое, но хочу проверить лишний раз.

*Область видимости:* Верно, Движок. Та же переменная, ничего не изменилось. Вот, держи.

*Движок:* Отлично. Передаю значение `a`, то есть 2, функции `log( . . )`.

...

## Упражнение

Проверьте, хорошо ли вы поняли материал. Возьмите на себя роль Движка и проведите «разговор» с Областью видимости:

```
function foo(a) {  
    var b = a;  
    return a + b;  
}  
var c = foo( 2 );
```

1. Найдите все LHS-поиски (всего 3).
2. Найдите все RHS-поиски (всего 4).



Ответы к упражнению приведены после раздела «Итоги» этой главы.

## Вложенная область видимости

Ранее было сказано, что область видимости — набор правил поиска переменных по имени идентификатора. Впрочем, обычно приходится принимать во внимание не одну область видимости, а несколько.

Подобно тому как блок или функция может вкладываться внутрь другого блока или функции, области видимости могут вкладываться в другие области видимости. Итак, если переменную не удастся найти в текущей области видимости, движок обращается к следующей внешней области видимости. Это продолжается до тех пор, пока не будет найдена искомая переменная или не будет достигнута внешняя (то есть глобальная) область видимости.

Пример:

```
function foo(a) {  
    console.log( a + b );
```

```
}  
  
var b = 2;  
  
foo( 2 ); // 4
```

RHS-ссылку для `b` не удастся разрешить внутри функции `foo`, но она может быть разрешена во внешней области видимости (в данном случае глобальной).

Итак, возвращаясь к разговору между Движком и Областью видимости, мы услышим следующее:

*Движок:* Эй, Область видимости, знаешь, что такое `b`? У меня тут RHS-ссылка.

*Область видимости:* Нет, впервые слышу про такое.

*Движок:* Эй, Область видимости за пределами `foo`... Э, да ты глобальная Область видимости? Ну и ладно. Знаешь, что такое `b`? У меня тут RHS-ссылка.

*Область видимости:* Ага, знаю. Вот, держи.

Простые правила проверки вложенных областей видимости: Движок начинает с текущей области видимости и ищет переменную в ней. Если поиск не дает результатов, Движок поднимается на один уровень вверх и т. д. При достижении внешней глобальной области видимости поиск прекращается независимо от того, была найдена переменная или нет.

## Метафоры

Чтобы наглядно представить процесс разрешения вложенных областей видимости, вообразите высокое здание (рис. 1.1).