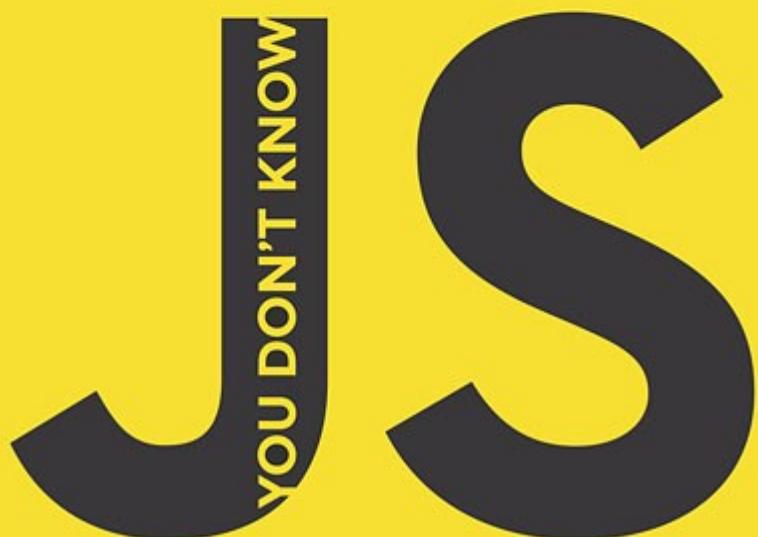


O'REILLY®

"Kyle's way of critically thinking about every tiny bit of the language will creep into your mindset and general workflow."  
—SHANE HUDSON, freelance frontend website developer

KYLE SIMPSON

# SCOPE & CLOSURES



# You Don't Know JS: *this* & Object Prototypes

## Table of Contents

- Foreword
- Preface
- Chapter 1: *this* Or That?
  - Why *this*?
  - Confusions
  - What's *this*?
- Chapter 2: *this* All Makes Sense Now!
  - Call-site
  - Nothing But Rules
  - Everything In Order
  - Binding Exceptions
  - Lexical *this*
- Chapter 3: Objects
  - Syntax
  - Type
  - Contents
  - Iteration
- Chapter 4: Mixing (Up) "Class" Objects
  - Class Theory
  - Class Mechanics
  - Class Inheritance
  - Mixins
- Chapter 5: Prototypes
  - `[[Prototype]]`
  - "Class"
  - "(Prototypal) Inheritance"
  - Object Links
- Chapter 6: Behavior Delegation
  - Towards Delegation-Oriented Design
  - Classes vs. Objects
  - Simpler Design
  - Nicer Syntax
  - Introspection
- Appendix A: ES6 `class`
- Appendix B: Acknowledgments

# Глава 1: Что такое область видимости?

Одна из самых фундаментальных парадигм почти всех языков программирования — возможность сохранять значения в переменных, а позже извлекать или менять эти значения. Собственно, возможность хранить значения и извлекать значения из переменных — это то, что дает программе *состояние*.

Без такого концепта программа могла бы выполнять некоторые задачи, но они были бы весьма ограничены и не были бы очень уж интересны.

Но включение переменных в нашу программу порождает самые интересные вопросы, которые мы теперь зададим: где эти переменные живут? Другими словами, где они хранятся? И, что более важно, как наша программа их находит, когда нуждается в них?

Эти вопросы говорят о необходимости хорошо определенного набора правил для хранения переменных в некоем месте и для обнаружения этих переменных позднее. Мы назовем этот набор правил — *Область видимости*.

Но где и как правила этих *областей видимости* устанавливаются?

## Теория компиляторов

Это может быть самоочевидно или это может удивлять, в зависимости от вашего уровня взаимодействия с различных языками, но несмотря на тот факт, что JavaScript подпадает под общую категорию "динамических" или "интерпретируемых" языков, на самом деле он является компилируемым языком. Он *не* компилируется заранее, как многие традиционно компилируемые языки, и результаты компиляции не являются переносимыми среди различных распределенных систем.

Но, тем ни менее, среда исполнения JavaScript выполняет много тех же шагов, что и любой традиционный компилятор языка, хоть и более сложными способами, чем мы обычно можем представить.

В традиционном процессе языковой компиляции, часть кода вашей программы обычно проходит три шага до того, как будет выполнена, в общих чертах называемых "компиляцией":

1. **Разбиение на лексемы (Tokenizing/Lexing)**: разбиение строки символов на имеющие смысл (для языка) части, называемые лексемами. Например, представьте программу: `var a = 2;`. Эта программа, вполне вероятно, будет разбита на следующие лексемы: `var`, `a`, `=`, `2` и `;`. Пробел может быть сохранен или не сохранен как лексема в зависимости от того имеет он смысл или нет.

**Примечание:** Разница между *tokenizing* и *lexing* — едва различима и теоретическая, но она сосредотачивается на том, идентифицируются ли эти лексемы как *без состояния* или *с состоянием*. Проще говоря, если токенизатор используется, чтобы вызывать правила парсинга с сохранением состояния для выяснения следует ли считать отдельной лексемой или только частью другой лексемы, это будет *lexing*.

**2. Парсинг:** берет поток (массив) лексем и превращает его в дерево вложенных элементов, которые сообща представляют грамматическую структуру программы. Это дерево называется "AST" (Abstract Syntax Tree, дерево абстрактного синтаксиса).

Такое дерево для `var a = 2;` может начинаться с узла верхнего уровня с названием `VariableDeclaration`, с дочерним узлом `Identifier` (чье значение равно `a`) и еще одним дочерним узлом `AssignmentExpression`, у которого тоже есть дочерний узел `NumericLiteral` (чье значение равно `2`).

**3. Генерация кода:** процесс взятия AST и превращения его в исполняемый код. Эта часть сильно зависит от языка, платформы назначения и т.п..

Итак, вместо того, чтобы увязать в деталях, мы просто опустим их и скажем, что есть способ взять наше вышеописанное AST для `var a = 2;` и превратить его в набор машинных инструкций, чтобы в действительности *создать* переменную с именем `a` (включая выделение памяти и т.д.), а затем *сохранить* значение в `a`.

**Примечание:** подробности того, как движок управляет системными ресурсами, глубже, чем мы будем "копать", поэтому мы всего лишь примем на веру, что движок умеет создавать и сохранять переменные, когда необходимо.

Движок JavaScript гораздо сложнее, чем *только* эти три шага, как и большинство других языковых компиляторов. Например, в процессе парсинга и генерации кода безусловно есть шаги по оптимизации быстродействия выполнения, включая удаление лишних элементов.

Поэтому здесь я лишь очерчиваю границы. Но я думаю, что вы скоро увидите почему эти детали, которые мы *обязательно* рассмотрим, хоть и на более высоком уровне, связаны.

Для начала, движки JavaScript не балуют роскошью (как другие языковые компиляторы) затрат массы времени на оптимизацию, так как компиляция JavaScript не происходит на шаге сборки заранее, как в других языках.

Для JavaScript, компиляция во многих случаях происходит за всего лишь микросекунды (или меньше!) перед выполнением кода. Чтобы гарантировать высочайшее быстродействие, движки JS используют все виды уловок (такие как JIT, который компилирует лениво и даже перекомпилирует на ходу), которые вне "области" нашего обсуждения тут.

Давайте скажем, простоты ради, что любой код JavaScript должен быть скомпилирован до (обычно *прямо перед!*) его выполнения. Поэтому, компилятор JS возьмет программу `var a = 2;` и *сперва* скомпилирует ее, а потом будет готов выполнить ее, обычно сразу же.

## › Понимание области видимости

---

Путь, которым мы будем приближаться к обучению области видимости, это думать о процессе в терминах диалога. Но, кто ведет этот диалог?

## › Действующие лица

Встречайте действующих лиц, которые взаимодействуют, чтобы обработать программу `var a = 2;`. Чтобы мы могли понять о чём их диалоги мы немного подслушаем их:

1. *Движок*: отвечает за компиляцию от начала до конца и выполнение нашей JavaScript программы.
2. *Компилятор*: один из друзей *Движка*, выполняет всю грязную работу по парсингу и генерации кода (см. предыдущий раздел).
3. *Область видимости*: еще один друг *Движка*; собирает и обслуживает список поиска всех объявленных идентификаторов (переменных), и следит за исполнением строгого набора правил о том, как эти идентификаторы доступны для кода, выполняемого в текущий момент.

Для *полного понимания* как работает JavaScript, вам необходимо начать думать как *Движок* (и его друзья), задавать вопросы, которые задают они и отвечать на них также.

## Туда и обратно

Когда вы видите программу `var a = 2;`, вы вероятнее всего подумаете о ней как об одном операторе. Но наш новый друг *Движок* видит это не так. На самом деле, *Движок* видит два отдельных оператора, один, который *Компилятор* обработает во время компиляции, а другой, который *Движок* обработает во время выполнения.

Так давайте же разберем по полочкам как *Движок* и его друзья поступят с программой `var a = 2;`.

Первая вещь, которую сделает *Компилятор* с этой программой, выполнит разбиение на лексемы, которые он затем распарсит в дерево. Но когда *Компилятор* доберется до генерации кода, он будет интерпретировать программу несколько по-другому нежели предполагалось.

Обоснованным предположением будет то, что *Компилятор* породит код, который можно кратко представить следующим псевдо-кодом: "Выделить память для переменной, пометить ее как `a`, затем поместить значение `2` в эту переменную." К сожалению, это не совсем точно.

*Компилятор* вместо этого сделает следующее:

1. Встретив `var a`, *Компилятор* просит *Область видимости* посмотреть существует ли уже переменная `a` в коллекции указанной области видимости. Если да, то *Компилятор* игнорирует это объявление переменной и двигается дальше. В противном случае, *Компилятор* просит *Область видимости* объявить новую переменную `a` в коллекции указанной области видимости.
2. Затем *Компилятор* генерирует код для *Движка* для последующего выполнения, чтобы обработать присваивание `a = 2`. Код, который *Движок* запускает, сначала спрашивает *Область видимости* есть ли переменная с именем `a`, доступная в коллекции текущей области видимости. Если да, то *Движок* использует эту переменную. Если нет, то *Движок* ищет в другом месте (см. раздел *Вложенная область видимости* ниже).

Если *Движок* в итоге находит переменную, он присваивает ей значение `2`. Если нет, то *Движок* вскинет руки и выкрикнет "ошибка!"!

Подводя итоги: для присваивания значения переменной выполняется два отдельных действия: первое, *Компилятор* обрабатывает переменную (если не была объявлена до этого в текущей области видимости), и второе, когда выполняет код, *Движок* ищет эту переменную в *Области видимости* и присваивает ей значение, если находит.

## ⁹ Компилятор расскажет

Нам нужно еще немного компиляторной терминологии, чтобы двинуться дальше.

Когда *Движок* выполняет код, который *Компилятор* генерирует на шаге (2), он должен поискать переменную *a*, чтобы увидеть была ли она объявлена и этот поиск принимает во внимание *Область видимости*. Но тип поиска, который выполняет *Движок*, влияет на результат поиска.

В нашем случае, он говорит, что *Движок* выполнит "LHS"-поиск переменной *a*. Другой тип поиска называется "RHS".

Держу пари, что вы можете угадать что значит "L" и "R". Эти термины означают "Left-hand Side" (левая сторона) и "Right-hand Side" (правая сторона).

Сторона... чего? **Операции присваивания.**

Иными словами, LHS-поиск выполняется, когда переменная появляется с левой стороны операции присваивания, а RHS-поиск выполняется, когда переменная появляется с правой стороны операции присваивания.

На самом деле, давайте будем более точны. RHS-поиск неотличим, для наших целей, от простого поиска значения некоторой переменной, тогда как LHS-поиск пытается найти сам контейнер переменной, чтобы он мог присвоить значение. Таким образом, RHS не обязательно означает "правая сторона присваивания" по существу, он просто более точно означает "не левая сторона".

Прикинувшись немного поверхностным на минуту, вы можете подумать, что "RHS" вместо этого значит "retrieve his/her source (value)" (получить его/ее исходное значение), представляя, что RHS означает "иди и возьми значение из...".

Давайте копнем немного глубже в этом направлении.

Когда я говорю:

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

Ссылка на *a* — это RHS-ссылка, потому что здесь ничего не присваивается в *a*. Напротив, мы выполняем поиск, чтобы извлечь значение *a*, для того, чтобы передать значение в `console.log(..)`.

Для сравнения:

```
a = 2;
```

Ссылка на а здесь — это LHS-ссылка, так как мы не заботимся здесь о том, каково текущее значение, мы просто хотим найти эту переменную как цель для операции присваивания = 2.

**Примечание:** LHS и RHS, означающие "левая/правая сторона присваивания", не обязательно буквально означают "левая/правая сторона операции присваивания =". Есть еще несколько способов, которыми производится присваивание, и поэтому лучше концептуально думать о нем как: "кто является целью присваивания (LHS)" и "кто источник присваивания (RHS)".

Представьте такую программу, в которой есть обе ссылки LHS и RHS:

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

foo( 2 );
```

Последняя строка, которая активизирует foo(..) как вызов функции, требует RHS-ссылку на foo, что значит, "сходи и найди значение foo и дай его мне". Более того,(..) означает, что значение foo должно быть выполнено, поэтому это скорее всего функция!

Здесь есть едва уловимое, но важное присваивание. **Вы обнаружили его?**

Вы наверное упустили неявное a = 2 в этом коде. Это происходит, когда значение 2 передается как аргумент в функцию foo(..), в этом случае значение 2 присваивается параметру a. Чтобы (неявно) присвоить значение параметру a, выполняется LHS-поиск.

Также есть и RHS-ссылка на значение a и это результирующее значение передается в console.log(..). console.log(..) нужна ссылка для выполнения. Для объекта console это RHS-поиск, затем происходит разрешение имени свойства чтобы убедиться существует ли метод, называемый log.

Наконец, мы можем осмыслить, что есть LHS/RHS-обмен передаваемым значением 2 (путем RHS-поиска переменной a) в log(..). Внутри родной реализации log(..), мы можем предположить, что у нее есть параметры, у первого из которых (возможно называющегося arg1) есть поиск LHS-ссылки, до присваивания ему 2.

**Примечание:** У вас может появиться соблазн представлять объявление функции function foo(a) {...} как обычное объявление переменной и присваивание, такое как var foo И foo = function(a){....}. Делая так, будет соблазн думать об объявлении этой функции как подразумевающей LHS-поиск.

Однако, едва заметная, но важная разница есть в том, что Компилятор обрабатывает как объявление, так и определение значения во время генерации кода, благодаря чему, когда Двигок выполняет код, не требуется никакой обработки чтобы "присвоить" значение функции в foo. Следовательно, неуместно думать об объявлении функции как о присваивании с помощью LHS-поиска тем способом, который мы здесь обсуждаем.

## › Беседа Двигка и Области видимости

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

foo( 2 );
```

Давайте представим вышеуказанный обмен между ними (который обрабатывает этот код) как беседу. Беседа может пойти примерно так:

**Двигок:** Эй, Область видимости, у меня есть RHS-ссылка на foo. Когда-нибудь слышала о такой?

**Область видимости:** Ну разумеется, слышала. Компилятор объявил ее всего секунду назад. Это функция. Пожалуйста!

**Двигок:** Отлично, спасибо! Хорошо, я выполняю foo.

**Двигок:** Эй, Область видимости, у меня есть LHS-ссылка на a, слышала что-нибудь о ней?

**Область видимости:** Ну разумеется, слышала. Компилятор объявил ее как формальный параметр в foo только что. Пожалуйста!

**Двигок:** Отзывчива как всегда, Область видимости. Снова спасибо. А теперь присвоим 2 в a.

**Двигок:** Эй, Область видимости, извини, что беспокою тебя снова. Мне нужен RHS-поиск console. Когда-нибудь слышала о таком имени?

**Область видимости:** Нет проблем, Двигок, это то, чем я весь день и занимаюсь. Да, у меня есть console. Она встроенная. Пожалуйста!

**Двигок:** Идеально. Ищу log(...). Превосходно, это функция.

**Двигок:** Эй, Область видимости. Можешь помочь мне с RHS-ссылкой на a? Думаю, я ее помню, но просто хочу лишний раз проверить.

**Область видимости:** Ты прав, Двигок. Та же ссылка, не изменилась. Пожалуйста!

**Двигок:** Круто! Передаю значение a, которое равно 2, в log(...).

...

## Тест

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

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

```
}
```

```
var c = foo( 2 );
```

1. Определите все LHS-поиски (их 3!).

2. Определите все RHS-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 области 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
```

RHS-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 RHS-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 RHS-ссылка на нее."

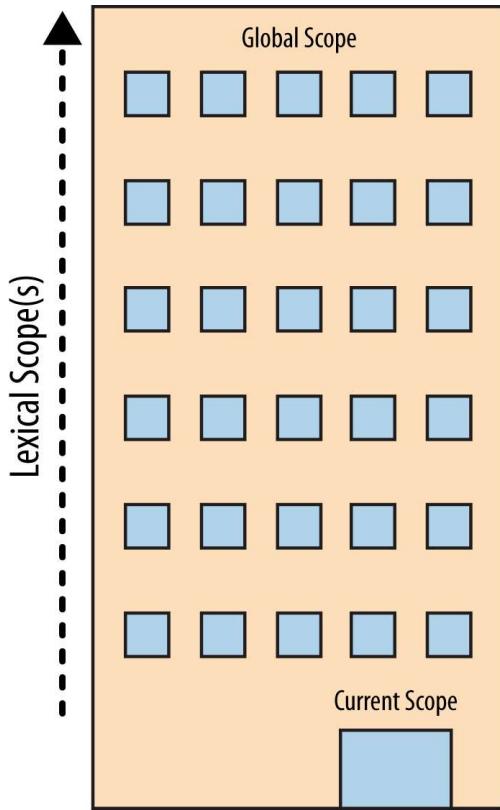
**Область видимости:** "Да-да, конечно есть. Пожалуйста!"

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.



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

Вы разрешаете LHS- и RHS-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 LHS или RHS?

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 RHS-поиск в 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 ReferenceError. Важно отметить, что эта ошибка имеет тип ReferenceError.

Напротив, если *Движок* выполняет LHS-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 LHS-поиска, и *Движок* выбросит ReferenceError аналогично случаю с RHS.

Теперь, если переменная найдена в ходе RHS-поиска, но вы пытаетесь сделать что-то с ее значением, что невозможно, например, пытаетесь выполнить как функцию нефункциональное значение или ссылаетесь на свойство значения null или undefined, то *Движок* выдаст другой вид ошибки, называемый TypeError.

ReferenceError — это сбой разрешения имени, связанный с *Областью видимости*, тогда как TypeError 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 LHS (left-hand-side) ссылкой, или может осуществляться для целей извлечения ее значения, которое является RHS (right-hand-side) ссылкой.

LHS-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 (LHS-ссылку) и присваивает ей значение, если находит.

Оба поиска ссылок LHS и RHS 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 RHS-ссылки приводят к выбросу `ReferenceError`. Невыполненные LHS-ссылки приводят к автоматической, неявносозданной переменной с таким именем (если не включен "Строгий режим"), либо к `ReferenceError` (если включен "Строгий режим").

## Ответы к тесту

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

1. Определите все LHS-поиски (их 3!).

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

2. Определите все RHS-поиски (их 4!).

`foo(2.., = a;, a + .. И .. + b`

Про "Строгий режим" [см. здесь](#)

# 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 А. 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 (lexing или tokenizing). Если вы припомните, то процесс разбиения на лексемы анализирует символы строки исходного кода и дает семантическое значение лексемам как результат некоторого парсинга 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.

```

function foo(a) {
  var b = a * 2;
  function bar(c) {
    console.log( a, b, c );
  }
  bar(b * 3);
}
foo( 2 ); // 2, 4, 12

```

1

2

3

**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 функций.

## 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 с `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 (shadowing)" (внутренний идентификатор "затеняет (shadows)" 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 есть 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, и b 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 ); // ReferenceError: a is not defined  
}  
  
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 генерированной 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 генерированную 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 генерированного 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 ); // undefined
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 `undefined`.

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 том 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 LHS-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 `with` подвергаются 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 *Двигок* 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 *Двигок* 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 *Двигка* выполнять оптимизации во время компиляции, принимающие во внимание поиск в области видимости, так как *Двигок* должен пессимистически предположить, что такие оптимизации будут неправильными. Код будет выполнятся медленнее в результате использования любой из этих возможностей. **Не используйте их!**

# Глава 3: Область видимости: функции против блоков

Как мы уже исследовали в главе 2, область видимости состоит из серии "зон", каждая из которых действует как контейнер или корзина, в которой объявляются идентификаторы (переменные, функции). Эти зоны четко вкладываются друг в друга и это вложение определяется во время написания кода.

Но что именно конкретно создает новую зону? Только функция? Могут ли другие структуры в JavaScript создавать зоны областей видимости?

## Область видимости из функций

Самый общий ответ на эти вопросы такой — в JavaScript есть области видимости в функциях. То есть, каждая функция, которую вы объявляете, создает зону для себя, но больше ни одна структура не создает свою собственную зону области видимости. Как мы скоро увидим, это не совсем правда.

Но сначала, давайте исследуем область видимости функции и ее применения.

Представьте такой код:

```
function foo(a) {  
    var b = 2;  
  
    // некоторый код  
  
    function bar() {  
        // ...  
    }  
  
    // еще код  
  
    var c = 3;  
}
```

В этом коде зона области видимости для `foo(..)` включает в себя идентификаторы `a`, `b`, `c` и `bar`. **Не важно где** в области видимости появится объявление, переменная или функция принадлежит к содержащей их зоне области видимости, вне зависимости от места объявления. Мы исследуем в следующей главе как это в точности работает.

У `bar(..)` есть своя собственная зона области видимости. Также и у глобальной области видимости, у которой есть всего один идентификатор: `foo`.

Так как `a`, `b`, `c` и `bar` все принадлежат к зоне области видимости `foo(..)`, они недоступны вне `foo(..)`. То есть, нижеприведенный код весь целиком приведет к ошибкам `ReferenceError`,

так как идентификаторы недоступны в глобальной области видимости:

```
bar(); // ошибка  
  
console.log( a, b, c ); // все 3 вызовут ошибку
```

Однако, все эти идентификаторы (`a`, `b`, `c`, `foo` и `bar`) доступны *внутри* `foo(..)` и конечно также доступны *внутри* `bar(..)` (предполагаем, что нет ни одного объявления затеняющих идентификаторов *внутри* `bar(..)`).

Область видимости функции поощряет идею, что все переменные принадлежат функции и могут быть использованы и переиспользованы на всем протяжении функции (и более того, доступны даже во вложенных областях видимости). Этот подход к дизайну может быть довольно полезен и несомненно может использовать в полном объеме "динамическую" природу переменных JavaScript, чтобы иметь дело со значениями разных типов при необходимости.

С другой стороны, если вы не принимаете меры предосторожности, переменные, существующие на всей протяженности области видимости, могут привести к неожиданным ошибкам.

## Прячемся на виду у всей области видимости

---

Традиционный путь рассуждений о функциях таков: вы определяете функцию, а затем добавляете код внутрь нее. Но обратное рассуждение столь же обоснованно и удобно: взять любую произвольную часть кода, которую вы написали, и обернуть ее в объявление функции, что в сущности "прячет" код.

Практическим результатом является создание зоны области видимости вокруг кода, о котором идет речь, а это значит, что любые объявления (переменной или функции) в этом коде будут ограничены областью видимости новой обворачивающей код функции, вместо ранее охватывающей области видимости. Иными словами, вы можете "спрятать" переменные и функции заключив их в область видимости функции.

Почему же "прятать" переменные и функции — полезная техника?

Есть множество причин, мотивирующих на это сокрытие, основанное на области видимости. Они имеют тенденцию проистекать из принципа дизайна ПО "Принцип наименьших привилегий", также иногда называемый "Наименьшие полномочия" или "Наименьшая открытость". Этот принцип заявляет, что в дизайне ПО, таком как API для модуля/объекта, вам следует выставлять наружу только то, что минимально необходимо и "прятать" всё остальное.

Этот принцип распространяется и на выбор того, какая область видимости будет содержать переменные и функции. Если все переменные и функции были бы в глобальной области видимости, они были бы, конечно, доступны любой вложенной области видимости. Но это нарушает принцип "Наименьшей..." в том, что вы (видимо) открываете многие переменные или функции, которые вам следовало в противном случае держать приватными, коль скоро правильное использование кода будет препятствовать доступу к этим переменным/функциям.

Например:

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

function doSomethingElse(a) {
    return a - 1;
}

var b;

doSomething( 2 ); // 15
```

В этом коде, переменная `b` и функция `doSomethingElse(..)` — похоже "частные" детали того, как `doSomething(..)` выполняет свою работу. Предоставление окружающей области видимости "доступа" к `b` и `doSomethingElse(..)` не только не необходимо, но также возможно и "опасно" тем, что их могут нечаянно использовать, намеренно или нет, и это может нарушить предварительные соглашения в `doSomething(..)`.

Более "правильный" дизайн скрыл бы эти частные детали внутри области видимости `doSomething(..)`, например как тут:

```
function doSomething(a) {
    function doSomethingElse(a) {
        return a - 1;
    }

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( (b * 3) );
}

doSomething( 2 ); // 15
```

Теперь, `b` и `doSomethingElse(..)` недоступны для любого внешнего воздействия, а взамен управляются только в `doSomething(..)`. Функциональность и конечный результат оказались не затронуты, а дизайн хранит частные детали частными, что обычно считается лучшим ПО.

## ‣ Предотвращение коллизий

Еще одно преимущество "скрытия" переменных и функций внутри области действия — чтобы избежать неумышленных коллизий между двумя идентификаторами с одним и тем же именем, но с разным целевым использованием. Коллизии часто приводят к неумышленной перезаписи значений.

Например:

```
function foo() {
    function bar(a) {
        i = 3; // меняем `i` в окружающей области видимости цикла for-loop
        console.log( a + i );
    }

    for (var i=0; i<10; i++) {
        bar( i * 2 ); // упс, впереди бесконечный цикл!
    }
}

foo();
```

Присваивание `i = 3` внутри `bar(..)` неожиданно перезаписывает `i`, которая была объявлена в `foo(..)` внутри цикла `for-loop`. В этом случае, это приведет к бесконечному циклу, так как `i` установлена в фиксированное значение 3 и она всегда будет оставаться `< 10`.

Присваиванию внутри `bar(..)` нужно объявить локальную переменную для своих нужд, независимо от того, какое имя идентификатора выбрано. `var i = 3;` исправило бы эту проблему (и создало бы ранее упоминавшееся объявление "затененной переменной" `i`). Дополнительная, не альтернативная возможность — выбрать совсем другое имя идентификатора, такое как `var j = 3;`. Но дизайн вашего ПО естественно может подразумевать использование одного и того же имени идентификатора, поэтому использование области видимости, чтобы "скрыть" ваше внутреннее объявление — это ваша лучшая/единственная возможность в этом случае.

## › Глобальные "Пространства имен"

Особенно яркий пример (наверное) коллизии переменных возникает в глобальной области видимости. Многие библиотеки, загружаемые в вашу программу, могут без всякого труда вступить в коллизию друг с другом, если они не спрячут должным образом свои внутренние/приватные функции и переменные.

Такие библиотеки, как правило, создают единственное объявление переменной, часто объекта, с достаточно уникальным именем в глобальной области видимости. Этот объект затем используется как "пространство имен" для этой библиотеки, где вся открываемая характерная функциональность делается как свойства этого объекта (пространства имен), вместо того, чтобы выставлять свои идентификаторы в области видимости на самом верхнем уровне.

Например:

```
var MyReallyCoolLibrary = {
    awesome: "stuff",
    doSomething: function() {
        // ...
    },
    doAnotherThing: function() {
```

```
// ...
}
};
```

## › Управление модулями

Еще одна возможность избежать коллизии — более современный "модульный" подход, используя любой из множества диспетчеров внедрения зависимостей. Используя эти инструменты, ни одна библиотека никогда не добавит ни одного идентификатора в глобальную область видимости, но взамен требуется явно импортировать их идентификатор в другую особую область видимости используя различные механизмы диспетчера внедрения зависимостей.

Следует отметить, что эти инструменты не обладают "волшебной" функциональностью, которая свободна от правил лексической области видимости. Они просто используют правила области видимости, о которых уже рассказывалось здесь, чтобы управлять тем, что ни один идентификатор не будет пущен ни в одну общую область видимости, а взамен идентификаторы будут храниться в приватных, не подверженных коллизиям областях видимости, которые предотвратят любые случайные коллизии областей видимости.

Собственно, вы можете кодировать осторожно и добиться таких же результатов как диспетчеры внедрения зависимостей без реальной необходимости их использования, если таков будет ваш выбор. Детальная информация о модульном шаблоне есть в главе 5.

## › Функции как области видимости

---

Мы уже видели, что можно взять любой кусок кода и обернуть его в функцию, и это эффективно "скроет" любые вложенные определения переменных или функций от внешней области видимости во внутренней области видимости этой функции.

Например:

```
var a = 2;

function foo() { // <-- вставляем это

    var a = 3;
    console.log( a ); // 3

} // <-- и это
foo(); // <-- и это

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

Несмотря на то, что эта техника "работает", она не обязательно очень идеальна. Она привносит некоторые проблемы. Первая, это то, что мы должны объявить именованную функцию `foo()`, которая означает, что само имя идентификатора `foo` "засоряет" окружающую область видимости (в данном случае глобальную). Мы также должны явно вызвать функцию по имени (`foo()`) для того, чтобы обернутый код выполнился на самом деле.

Было бы идеально, если бы функции не было нужно имя (или точнее, чтобы это имя не загрязняло окружающую область видимости) и если бы функция могла бы автоматически выполняться.

К счастью, JavaScript предлагает решение обеих проблем.

```
var a = 2;

(function foo(){ // <-- вставляем это

    var a = 3;
    console.log( a ); // 3

})(); // <-- и это

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

Давайте разберем в деталях что же тут происходит.

Сперва, заметьте, что оператор окружающей функции начинается с `(function...` в противоположность простому `function....`. Хоть это и может показаться несущественной деталью, это на самом деле огромное изменение. Вместо того, чтобы трактовать функцию как стандартное объявление, функция трактуется как функциональное выражение.

**Примечание:** Самый легкий путь отличить объявление от выражения — позиция слова "function" в операторе (не только строка, но и отдельный оператор). Если "function" — самое первое, что стоит в операторе, то это объявление функции. Иначе, это функциональное выражение.

Ключевое отличие между объявлением функции и функциональным выражением, которое мы можем заметить тут, связано с тем, где его имя связывается с идентификатором.

Сравните два предыдущих кода. В первом, имя `foo` связано с окружающей областью видимости и мы вызываем его напрямую через `foo()`. Во втором коде, имя `foo` не связано с окружающей областью видимости, а заменено связанным только со своей собственной функцией.

Иными словами, `(function foo(){ .. })` как выражение означает, что идентификатор `foo` может быть найден только в области видимости, которая обозначена `..`, но не во внешней области видимости. Скрытие имени `foo` внутри себя означает, что оно не будет неоправданно загрязнять окружающую область видимости.

## ‣ Анонимный против названного

Возможно, вы лучше всех знакомы с функциональными выражениями как параметрами функций обратного вызова, например:

```
setTimeout( function(){
    console.log("I waited 1 second!");
}, 1000 );
```

Это называется "анонимное функциональное выражение", так как у `function()...` нет именованного идентификатора. Функциональные выражения могут быть анонимными, но объявления функций не могут опускать имя — это было бы невалидным синтаксисом JS.

Анонимные функциональные выражения быстро и легко вводить и многие библиотеки и утилиты проявляют тенденцию к поощрению этого идиоматического стиля кода. Однако, у них есть несколько недостатков о которых нужно упомянуть:

1. У анонимных функций нет удобного имени для отображения в стектрейсах (stacktrace), что может затруднить отладку.
2. Если функции без имени будет нужно сослаться на себя же, для рекурсии или чего-то подобного, к сожалению требуется устаревшая ссылка `arguments.callee`. Еще один пример необходимости в ссылке на себя, когда функция обработчика события хочет отписать себя от события после выполнения.
3. Анонимные функции опускают имя, что часто удобно для обеспечения большей читаемости/понятности кода. Наглядное же имя помогает самодокументировать рассматриваемый код.

**Встраиваемые функциональные выражения** — мощные и полезные инструменты, выбор между анонимными и именованными не умаляет их достоинств. Именование вашего функционального выражения достаточно эффективно решает все перечисленные недостатки, и при этом не имеет ощутимых минусов. Лучшая практика — это всегда именовать ваши функциональные выражения:

```
setTimeout( function timeoutHandler(){ // <-- Смотри, у меня есть имя!
    console.log( "I waited 1 second!" );
}, 1000 );
```

## › Вызов функциональных выражений по месту

```
var a = 2;

(function foo(){
    var a = 3;
    console.log( a ); // 3

})();

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

Теперь, когда у нас есть функция как выражение благодаря обертыванию ее в пару `( )`, мы можем вызвать эту функцию добавив еще одни `()` в конце, как тут `(function foo(){ .. })(())`. Первая окружающая пара `( )` делает функцию выражением, а вторая `()` выполняет функцию.

Этот шаблон настолько в ходу, что несколько лет назад сообщество условилось о термине для него: **IIFE**, что означает **Immediately (немедленно) Invoked (вызываемое) Function**

## (функциональное) Expression (выражение).

Конечно, IIFE не обязательно нужны имена, самая распространенная форма IIFE — использование анонимного функционального выражения. Несмотря на то, что определенно менее используемое, именование IIFE имеет все вышеперечисленные преимущества над анонимными функциональными выражениями, поэтому взять его на вооружение — хорошая практика.

```
var a = 2;

(function IIFE(){

    var a = 3;
    console.log( a ); // 3

})();

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

Есть легкая вариация формы традиционной IIFE, которую предпочитают некоторые: `(function(){ .. }())`. Посмотрим ближе, чтобы увидеть разницу. В первой форме функциональное выражение обернуто в `( )`, а затем пара вызывающих ее `()` снаружи прямо за ними. Во второй форме,зывающая пара `()` переместилась внутрь внешней окружающей пары `( )`.

Эти две формы идентичны по функциональности. **Какую из них предпочесть — всего лишь ваш стилистический выбор.**

Еще одна вариация IIFE, которая широко распространена, это использование того факта, что они, на самом деле, просто вызовы функций, и передача им аргумента(ов).

Например:

```
var a = 2;

(function IIFE( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

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

Мы передаем ссылку на объект `window`, но параметр мы назвали `global`, так что у нас есть ясное стилистическое разграничение для глобальных и неглобальных ссылок. Конечно, вы можете передать внутрь что-то из окружающей области видимости, если хотите, и можете назвать параметр(ы) так, как удобно вам. Это в основном всего лишь стилистический выбор.

Еще одно применение этого шаблона решает (узкоспециализированную) проблему, что значение идентификатора по умолчанию `undefined` может быть некорректно перезаписано, приведя к неожиданным результатам. Давая параметру имя `undefined`, но не передавая никакого значения для этого аргумента, мы можем гарантировать, что идентификатор `undefined` фактически имеет незаданное значение в этом блоке кода:

```
undefined = true; // устанавливаем мину для другого кода! остерегайтесь!

(function IIFE( undefined ){
    var a;
    if (a === undefined) {
        console.log( "Undefined is safe here!" );
    }
})();
```

Еще одна вариация IIFE меняет порядок вещей на обратный, где вызываемая функция идет второй, *после* вызова и параметров, которые в нее передаются. Этот шаблон используется в проекте UMD (Universal Module Definition). Некоторые люди находят его более ясным для понимания, хотя он немного более многословный.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2
});
```

Функциональное выражение `def` определяется во второй половине кода, а затем передается как параметр (также названный `def`) в функцию IIFE, определенную в первой половине кода. Наконец, параметр `def` (функция) вызывается, передавая `window` в нее как параметр `global`.

## Блоки как области видимости

В то время как функции являются самыми распространенными единицами области видимости и определенно самые распространенные подходы к разработке в большей части JS, есть и другие единицы области видимости, и использование этих единиц области видимости может привести к даже лучшему, более чистому управлению кодом.

Многие языки, отличные от JavaScript, поддерживают блочную область видимости и поэтому разработчики из этих языков привыкли к такому мышлению, в свою очередь те, кто изначально работали в JavaScript могут найти эту концепцию немного чуждой.

Но даже если вы никогда не писали ни строчки кода в стиле блочной области видимости, вы возможно все-таки знакомы с этой чрезвычайно общей идиомой в JavaScript:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

Мы объявляем переменную `i` прямо внутри заголовка цикла `for-loop`, скорее всего потому, что наше *намерение* — использовать `i` только в контексте этого цикла `for-loop` и в основном игнорировать факт того, что переменная на самом деле заключает себя в окружающую область видимости (функции или глобальную).

Вот это и есть самое важное в блочной области видимости. Объявление переменных как можно ближе и как можно локальней к тому месту, где они будут использоваться. Еще один пример:

```
var foo = true;

if (foo) {
    var bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}
```

Мы используем переменную `bar` только в контексте оператора `if`, поэтому вполне разумно, что мы объявили ее внутри блока `if`. Однако, место, где мы объявляем переменные не имеет значения при использовании `var`, так как они всегда принадлежат окружающей области видимости. Этот код по существу "поддельная" блочная область видимости, только для целей стилистики, и полагается на самообязательство не использовать случайно `bar` в другом месте этой области видимости.

Блочная область видимости — это инструмент для расширения ранее упоминаемого "Принципа наименьшей привилегии открытости" с сокрытия информации в функциях на сокрытие информации в блоках вашего кода.

Давайте еще раз посмотрим пример с `for-loop`:

```
for (var i=0; i<10; i++) {
    console.log( i );
}
```

Зачем загрязнять всю область видимости функции переменной `i`, которая будет использоваться (или только следует, по меньшей мере) в цикле `for-loop`?

Но более важно, что разработчики могут предпочесть проверить себя на предмет случайного (пере)использования переменных снаружи от их предполагаемой области действия, как например получить ошибку о неизвестной переменной, если вы пытаетесь использовать ее не в том месте. Блочная область видимости (если бы она была возможна)

для переменной `i` сделала бы `i` доступной только для цикла `for-loop`, приводя к ошибке, если к `i` осуществляется доступ в другом месте функции. Это помогает убедиться, что переменные не будут переиспользованы в нечетких или труднообслуживаемых сценариях.

Но, печальная реальность состоит в том, что, на первый взгляд, в JavaScript нет возможности блочной области видимости.

Или, если точнее, пока вы не копнете немного глубже.

## › **with**

Мы изучали `with` в главе 2. Несмотря на то, что к нему относятся с неодобрением с момента его появления, оно является примером (формой) блочной области видимости, поскольку область видимости, которая создается из объекта, существует только в течение жизненного цикла этого оператора `with`, а не окружающей области видимости.

## › **try/catch**

Очень малоизвестным фактом является то, что JavaScript в ES3 специфицирует объявление переменной в блоке `catch` оператора `try/catch` как принадлежащее блочной области видимости блока `catch`.

Например:

```
try {
    undefined(); // нелегальная операция, чтобы вызвать исключение!
}
catch (err) {
    console.log( err ); // работает!
}

console.log( err ); // ReferenceError: `err` not found
```

Как видите, `err` существует только в блоке `catch` и выбрасывает ошибку когда вы пытаетесь сослаться на нее где-либо в другом месте.

**Примечание:** Несмотря на то, что такое поведение было определено и истинно практически во всех стандартных средах JS (за исключением разве что старого IE), многие линтеры (средства контроля качества кода) похоже до сих пор жалуются, если у вас есть два или более блоков `catch` в одной и той же области видимости, каждый из которых объявляет свою переменную ошибки с одинаковым именем идентификатора. В действительности это не переопределение, поскольку переменные безопасно обернуты в блочные области видимости, но линтеры похоже до сих пор раздражают на этот факт.

Чтобы избежать таких ненужных предупреждений, некоторые разработчики именуют свои переменные в `catch` как `err1`, `err2` и т.д.. Другие разработчики просто выключают проверку линтера на повторяющиеся имена переменных.

Природа блочной области видимости `catch` может казаться бесполезным академическим фактом, но лучше загляните в приложение B, чтобы получить детальную информацию о том

насколько полезна она может быть.

## › **let**

До сих пор мы видели, что в JavaScript есть только несколько странных нишевых возможностей, которые предлагают функциональность блочной области видимости. Если бы это было всё, что у нас есть, а так и было на протяжении многих, многих лет, то блочная область видимости не была бы столь полезна для разработчика на JavaScript.

К счастью, ES6 поменял ситуацию и представляет новое ключевое слово `let`, которое соседствует с `var` как еще один путь объявления переменных.

Ключевое слово `let` присоединяет объявление переменной к области видимости того блока (обычно пара `{ .. }`), в котором оно содержится. Иными словами, `let` неявно похищает у любой блочной области видимости ее объявления переменных.

```
var foo = true;

if (foo) {
    let bar = foo * 2;
    bar = something( bar );
    console.log( bar );
}

console.log( bar ); // ReferenceError
```

Использование `let` для присоединения переменной к существующему блоку в некоторой степени неявно. Оно может ввести в заблуждение, если вы не оказываете пристальное внимание на то, в каких блоках есть переменные, принадлежащие к их области видимости, и на привычку перемещать блоки, оборачивать их в другие блоки, и т.п., по мере того, как вы разрабатываете и развиваете код.

Создание явных блоков для блочной области видимости может решить некоторые из этих вопросов, делая более очевидным то, куда присоединены переменные, а куда — нет. Обычно, явный код предпочтительней неявного или едва заметного. Такой явный стиль блочной области видимости легко получить и он более естественно походит на то, как в других языках работает блочная область видимости:

```
var foo = true;

if (foo) {
    { // <-- явный блок
        let bar = foo * 2;
        bar = something( bar );
        console.log( bar );
    }
}

console.log( bar ); // ReferenceError
```

Можно создать обычный блок для использования `let` просто включая пару `{ .. }` в любом месте, где этот оператор является валидным синтаксисом. В этом случае, мы сделали явный блок *внутри* оператора `if`, который потом будет легче перемещать как целый блок при рефакторинге, без изменения позиции и семантики окружающего оператора `if`.

**Примечание:** Еще один путь объявить явные блочныe области видимости есть в приложении B.

В главе 4 мы рассмотрим поднятие переменных (*hoisting*), которая расскажет об объявлении, которые воспринимаются как существующие для всей области видимости, в которой они появляются.

Однако, объявления, сделанные с помощью `let`, не поднимаются во всей области видимости блока, в котором они появляются. Такие объявления очевидно не будут "существовать" в блоке до оператора объявления.

```
{
  console.log( bar ); // ReferenceError!
  let bar = 2;
}
```

## › Сборка мусора

Еще одна причина полезности блочной области видимости связана с замыканиями и сборкой мусора, чтобы освободить память. Мы кратко проиллюстрируем это здесь, но детально механизм замыканий будет рассматриваться в главе 5.

Пример:

```
function process(data) {
  // делаем что-то интересное
}

var someReallyBigData = { .. };

process( someReallyBigData );

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
  console.log("button clicked");
}, /*capturingPhase=*/false );
```

Обратный вызов обработчика щелчка `click` совсем не требует переменную `someReallyBigData`. Это значит, теоретически, что после выполнения `process(..)`, большая памятзатратная структура данных может быть собрана сборщиком мусора. Однако, весьма вероятно (хотя зависит от реализации), что движок JS все еще должен будет оставить структуру в памяти, поскольку у функции `click` есть замыкание, действующее во всей области видимости.

Блочная область видимости может устранить этот недостаток, делая более явным для движка то, что ему не нужна `someReallyBigData`:

```
function process(data) {
    // делаем что-то интересное
}

// всё, что объявлено внутри этого блока, может исчезнуть после него!
{
    let someReallyBigData = { .. };

    process( someReallyBigData );
}

var btn = document.getElementById( "my_button" );

btn.addEventListener( "click", function click(evt){
    console.log("button clicked");
}, /*capturingPhase=*/false );
```

Объявление явных блоков для переменных, чтобы локально привязать их к блокам — мощный инструмент, который вы можете добавить в свой арсенал.

## › `let` в циклах

Особый случай, в котором `let` показывает себя с лучшей стороны — в случае с циклом `for`, как мы уже обсуждали ранее.

```
for (let i=0; i<10; i++) {
    console.log( i );
}

console.log( i ); // ReferenceError
```

`let` в заголовке цикла `for` не только привязывает `i` к телу цикла, но фактически, он **перепривязывает ее** в каждой итерации цикла, обязательно переприсваивая ей значение с окончания предыдущей итерации.

Вот еще один способ показать режим пред-итеративной привязки:

```
{
    let j;
    for (j=0; j<10; j++) {
        let i = j; // перепривязка в каждой итерации!
        console.log( i );
    }
}
```

Причина, по которой эта пред-итеративная привязка интересна, станет ясна в главе 5, когда мы обсудим замыкания.

Так как объявления `let` присоединяются к обычным блокам вместо присоединения к окружающей области видимости функции (или глобальной), могут быть ошибки в программе, когда у существующего кода есть скрытая надежда на объявления `var`, действующие в рамках области видимости функции, и замена `var` на `let` может потребовать дополнительного внимания при рефакторинге кода.

Пример:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    if (baz > bar) {
        console.log( baz );
    }

    // ...
}
```

Этот код довольно легко отрефакторить в такой:

```
var foo = true, baz = 10;

if (foo) {
    var bar = 3;

    // ...

if (baz > bar) {
    console.log( baz );
}
```

Но, остерегайтесь таких изменений, когда используете переменные блочной области видимости:

```
var foo = true, baz = 10;

if (foo) {
    let bar = 3;

    if (baz > bar) { // <-- не забудьте про `bar` , если будете перемещать этот блок!
        console.log( baz );
    }
}
```

См. приложение В, чтобы узнать об альтернативном (более явном) стиле организации блочной области видимости, который может дать более простое обслуживание/рефакторинг кода, что более разумно для этих сценариев.

## › const

В дополнение к `let`, ES6 представляет ключевое слово `const`, которое также создает переменную блочной области видимости, но чье значение фиксированно (константа). Любая попытка изменить это значение позже приведет к ошибке.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // в блочной области видимости содержащего ее `if`

    a = 3; // просто отлично!
    b = 4; // ошибка!

}

console.log( a ); // 3
console.log( b ); // ReferenceError!
```

## › Обзор

---

Функции — самые распространенные единицы области видимости в JavaScript. Переменные и функции, которые объявляются внутри другой функции, по существу "скрыты" от любой из окружающих "областей видимости", что является намеренным принципом разработки хорошего ПО.

Но функции — отнюдь не только единицы области видимости. Блочная область видимости ссылается на идею, что переменные и функции могут принадлежать произвольному блоку (обычно, любой паре `{ .. }`) кода, нежели только окружающей функции.

Начиная с ES3, в структуре `try/catch` есть блочная область видимости в выражении `catch`.

В ES6 представлено ключевое слово `let` (родственница ключевого слова `var`), чтобы позволить объявления переменных в любом произвольном блоке кода. `if (..) { let a = 2;` `} объявит переменную a, которая фактически похитит область видимости блока { .. } в if и присоединит себя к ней.`

Хоть некоторые похоже и верят в это, но блочную область видимости не следует использовать как полную замену функциональной области видимости `var`. Обе функциональности сосуществуют вместе, а разработчики могут и должны использовать обе техники области видимости: функциональную и блочную, в соответствующих местах, чтобы создавать лучший, более читаемый/обслуживаемый код.

### Принцип наименьших привилегий

# Глава 4: Поднятие переменных (Hoisting)

Теперь вы должно быть вполне уверенно разбираетесь в идее области видимости, и в том, как переменные присоединяются к различным уровням области видимости в зависимости от того, где и как они объявлены. Обе области видимости, как функции, так и блока работают по одинаковым правилам в таком виде: любая переменная, объявленная в области видимости, присоединяется к этой области видимости.

Но есть маленький нюанс в том, как работает присоединение к области видимости с объявлениями, которые появляются в различных местах области видимости, и это именно тот нюанс, который мы тут и исследуем.

## Курица или яйцо?

Есть искушение подумать, что весь код, который вы видите в программе на JavaScript, интерпретируется строка за строкой. Несмотря на то, что по сути это правда, есть одна часть этого предположения, которая может привести к некорректному пониманию вашей программы.

Представьте такой код:

```
a = 2;  
  
var a;  
  
console.log( a );
```

Что вы ожидаете увидеть в выводе оператора `console.log(..)`?

Многие разработчики ожидают увидеть `undefined`, поскольку оператор `var a` идет после `a = 2`, и было бы естественным предположить, что переменная переопределена и потому ей присвоено значение по умолчанию `undefined`. Однако, результат будет 2.

Представьте еще один код:

```
console.log( a );  
  
var a = 2;
```

Вы можете склониться к предположению, что поскольку в предыдущем показанном коде есть некоторое поведение в стиле "немного с ног на голову", то возможно в этом коде также будет выведено 2. Другие могут подумать, что поскольку переменная `a` используется раньше, чем объявлена, то это приведет к выбросу `ReferenceError`.

К сожалению, оба предположения неверны. Будет выведено `undefined`.

Так что же здесь происходит? Похоже тут вопрос сродни "что раньше: курица или яйцо?". Что идет первым: объявление ("яйцо") или присваивание ("курица")?

## Компилятор снова наносит удар

Чтобы ответить на этот вопрос, нам нужно вернуться в главу 1 и нашу дискуссию о компиляторах. Вспомните, что Двигок на самом деле скомпилирует ваш код JavaScript до того, как начнет интерпретировать его. Частью фазы компиляции является нахождение и ассоциация всех объявлений с их соответствующими областями видимости. Глава 2 показала нам, что это и есть сердце лексической области видимости.

Поэтому, лучший путь думать об этих вещах — что все объявления как переменных, так и функций, обрабатываются в первую очередь, до того как будет выполнена любая часть вашего кода.

Когда видите `var a = 2;`, вы наверное думаете о нем как об одном операторе. Но JavaScript на самом деле думает о нем как о двух операторах: `var a;` и `a = 2;`. Первый оператор, объявление, обрабатывается во время фазы компиляции. Второй оператор, присваивание, остается **на своем месте** в фазе исполнения.

Следовательно о нашем первом коде следует думать как об обрабатываемом следующим образом:

```
var a;  
  
a = 2;  
  
console.log( a );
```

...где первая часть — компиляция, а вторая — выполнение.

Аналогично, наш второй код в действительности будет обработан так:

```
var a;  
  
console.log( a );  
  
a = 2;
```

Получается, один из путей представить это, в какой-то степени образно, что эти объявления переменной и функции "переехали" с того места, где они появились в коде в начало кода. Это дало начало названию "Поднятие (Hoisting)".

Другими словами, яйцо (объявление) появилось до курицы (присваивания).

**Примечание:** Поднимаются только сами объявления, тогда как любые присваивания или другая логика выполнения остается *на месте*. Если поднятие намеренно используется, чтобы перестроить логику выполнения вашего кода, то это может вызвать хаос.

```
foo();

function foo() {
    console.log( a ); // undefined

    var a = 2;
}
```

Объявление функции `foo` (которое в этом случае *включает в себя* соответствующее значение как актуальную функцию) поднимается, благодаря чему ее вызов в первой строке может быть выполнен.

Также важно отметить, что каждое поднятие соотносится с **областью видимости**. Поэтому несмотря на то, что наши предыдущие примеры кода были упрощенными в том, что были включены только в глобальную область видимости, функция `foo(..)`, которую мы сейчас изучаем, показывает, что `var a` поднимается наверх `foo(..)` (а не, очевидно, наверх всей программы). Так что программу можно было бы прочесть более точно как:

```
function foo() {
    var a;

    console.log( a ); // undefined

    a = 2;
}

foo();
```

Объявления функций поднимаются, как мы только что видели. А функциональные выражения — нет.

```
foo(); // не ReferenceError, но TypeError!

var foo = function bar() {
    // ...
};
```

Идентификатор переменной `foo` поднимается и присоединяется к включающей его области видимости (глобальной) этой программы, поэтому `foo()` не вызовет ошибки `ReferenceError`. Но в `foo` пока еще нет значения (как если бы это было объявление обычной функции вместо выражения). Поэтому, `foo()` пытается вызвать значение `undefined`, которое является неправильной операцией с вызовом ошибки `TypeError`.

Также помните, что даже если это именованное функциональное выражение, идентификатор имени недоступен в окружающей области видимости:

```
foo(); // TypeError
bar(); // ReferenceError

var foo = function bar() {
    // ...
};
```

Этот код более точно интерпретируется (с учетом поднятия) как:

```
var foo;

foo(); // TypeError
bar(); // ReferenceError

foo = function() {
    var bar = ...self...
    // ...
}
```

## Сначала функции

---

Как объявления функций, так и переменных поднимаются. Но тонкость (которая *поможет* объяснить множественные объявления "дубликатов" в коде) в том, что сперва поднимаются функции, а затем уже переменные.

Представим:

```
foo(); // 1

var foo;

function foo() {
    console.log( 1 );
}

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

1 выводится вместо 2! Этот код интерпретируется *Движком* так:

```
function foo() {
    console.log( 1 );
}

foo(); // 1

foo = function() {
```

```
        console.log( 2 );
    };
```

Обратите внимание, что `var foo` является дублем объявления (и поэтому игнорируется), даже несмотря на то, что идет до объявления `function foo()...`, потому что объявления функций поднимаются до обычных переменных.

В то время как множественные/дублирующие объявления `var` фактически игнорируются, последовательные объявления функции *перекрывают* предыдущие.

```
foo(); // 3

function foo() {
    console.log( 1 );
}

var foo = function() {
    console.log( 2 );
};

function foo() {
    console.log( 3 );
}
```

Несмотря на то, что всё это может прозвучать как не более чем любопытный академический факт, это привлекает тем, что дубли определений в одной и той же области видимости — в самом деле плохая идея и часто приводит к странным результатам.

Объявления функций, которые появляются внутри обычных блоков, обычно поднимаются в окружающей области видимости, вместо того чтобы быть условными как показывает код ниже:

```
foo(); // "b"

var a = true;
if (a) {
    function foo() { console.log( "a" ); }
}
else {
    function foo() { console.log( "b" ); }
}
```

Однако, важно отметить, что такое поведение небезопасно и может измениться в будущих версиях JavaScript, поэтому лучше избегать объявления функций в блоках.

## › Обзор

---

У нас есть соблазн смотреть на `var a = 2;` как на один оператор, но Двигок JavaScript видит это по-другому. Он видит `var a` и `a = 2` как два отдельных оператора, первый — как задачу

фазы компиляции, а второй — как задачу фазы выполнения.

Это приводит к тому, что все объявления в области видимости, независимо от того где они появляются, обрабатываются *первыми*, до того, как сам код будет выполнен. Можно мысленно представить себе это как объявления (переменных и функций), "переезжающие" в начало их соответствующих областей видимости, что мы называем "поднятие (hoisting)".

Сами объявления поднимаются, а присваивания, даже присваивания функциональных выражений, *не* поднимаются.

Остерегайтесь дублей объявлений, особенно смешанных обычных объявлений var и объявлений функций — вас будут поджидать неприятности!

```

        return {
            identify: identify
        };
    }

var foo1 = CoolModule( "foo 1" );
var foo2 = CoolModule( "foo 2" );

foo1.identify(); // "foo 1"
foo2.identify(); // "foo 2"

```

Еще одна небольшая, но полнофункциональная вариация модульного шаблона — дать имя объекту, который вы возвращаете как публичное API:

```

var foo = (function CoolModule(id) {
    function change() {
        // modifying the public API
        publicAPI.identify = identify2;
    }

    function identify1() {
        console.log( id );
    }

    function identify2() {
        console.log( id.toUpperCase() );
    }

    var publicAPI = {
        change: change,
        identify: identify1
    };

    return publicAPI;
})( "foo module" );

foo.identify(); // foo module
foo.change();
foo.identify(); // FOO MODULE

```

Сохраняя внутреннюю ссылку на объект публичного API внутри экземпляра модуля вы можете менять эту ссылку на модуль **изнутри**, включая добавление и удаление методов, свойств и изменение их значений.

## › Современные модули

Различные загрузчики/менеджеры модульных зависимостей фактически упаковывают это определение модульного шаблона в дружественное API. Прежде чем начать изучать любую

отдельную библиотеку, позвольте мне показать очень простой экспериментальный вариант **только в иллюстративных целях**:

```
var MyModules = (function Manager() {
    var modules = {};

    function define(name, deps, impl) {
        for (var i=0; i<deps.length; i++) {
            deps[i] = modules[deps[i]];
        }
        modules[name] = impl.apply(impl, deps);
    }

    function get(name) {
        return modules[name];
    }

    return {
        define: define,
        get: get
    };
})();
```

Ключевая часть этого кода — `modules[name] = impl.apply(impl, deps)`. Это вызов функции-обертки определения для модуля (с передачей внутрь любых зависимостей) и сохранение возвращаемого значения, API модуля, во внутренний список модулей со ссылкой по имени.

И вот как именно я мог бы его использовать для определения нескольких модулей:

```
MyModules.define( "bar", [], function(){
    function hello(who) {
        return "Let me introduce: " + who;
    }

    return {
        hello: hello
    };
} );

MyModules.define( "foo", ["bar"], function(bar){
    var hungry = "hippo";

    function awesome() {
        console.log( bar.hello( hungry ).toUpperCase() );
    }

    return {
        awesome: awesome
    };
} );

var bar = MyModules.get( "bar" );
```

```
var foo = MyModules.get( "foo" );

console.log(
  bar.hello( "hippo" )
); // Let me introduce: hippo

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

Оба модуля "foo" и "bar" объявлены с функцией, которая возвращает публичное API. "foo" даже получает экземпляр "bar" как параметр-зависимость и может его использовать соответствующим образом.

Потратите немного времени на изучение этих примеров кода, чтобы полностью осознать всю мощь замыканий и потом воспользоваться ими в собственных целях. Ключевой вывод тут в том, что нет никакого особого "волшебства" в модульных менеджерах. Они соответствуют обеим характеристикам шаблона модулей, которые я перечислил выше: вызов обертки определения функции и хранение возвращаемого значения в качестве API для этого модуля.

Иными словами, модули — это просто модули, даже если вы делаете ставку на хорошо знакомый вам инструмент для этого.

## › Будущие модули

ES6 добавляет синтаксическую поддержку первого класса для концепции модулей. При загрузке через модульную систему, ES6 обрабатывает файл как отдельный модуль. Каждый модуль может как импортировать другие модули или отдельные члены из API, так и экспортить свои собственные публичные члены API.

**Примечание:** Модули, основанные на функциях — не статически распознаваемый шаблон (о котором знает компилятор), поэтому их семантика API не учитывается до момента времени выполнения. То есть, вы фактически можете менять API модуля в процессе выполнения кода (см. ранее обсуждение публичного API).

В противоположность этому, модульное API ES6 — статическое (API не меняется во время выполнения). Поскольку компилятор это знает, он может (и делает!) проверить во время (загрузки файла и) компиляции, что ссылка на член API импортируемого модуля *действительно существует*. Если ссылка на API не существует, компилятор выдаст "заблаговременную" ошибку на этапе компиляции, вместо того, чтобы ждать традиционного динамического разрешения ссылок во время выполнения (и ошибок, если есть).

У модулей ES6 нет "встраиваемого" формата, они должны определяться в отдельных файлах (по одному на модуль). У браузеров/движков есть "загрузчик модулей" по умолчанию (который может быть переопределен, но это уже далеко за пределами нашего здесь обсуждения), который синхронно загружает файл модуля, когда он импортируется.

Рассмотрим код:

### bar.js

```
function hello(who) {
  return "Let me introduce: " + who;
```

}

```
export hello;
```

## foo.js

```
// импортирует только `hello()` из модуля "bar"
import hello from "bar";

var hungry = "hippo";

function awesome() {
    console.log(
        hello( hungry ).toUpperCase()
    );
}

export awesome;
```

```
// импортирует модули "foo" и "bar" целиком
module foo from "foo";
module bar from "bar";

console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

foo.awesome(); // LET ME INTRODUCE: HIPPO
```

**Примечание:** Необходимо создать отдельные файлы "foo.js" и "bar.js", с указанным выше содержимым. Затем ваша программа загрузит/импортирует эти модули, чтобы использовать их как показано в третьем фрагменте кода.

`import` импортирует один или более членов API модуля в текущую область видимости, каждый в свою переменную (`hello` в нашем случае). `module` импортирует API модуля целиком в указанную переменную (`foo, bar` в нашем случае). `export` экспортирует идентификатор (переменную, функцию) в публичное API текущего модуля. Эти операторы можно использовать в определении модуля столько раз, сколько потребуется.

Содержимое внутри *файла модуля* обрабатывается как если бы оно было заключено в замыкание области видимости, также как и у модулей с функцией-замыканием, рассмотренных ранее.

## Итог

---

Похоже, что знания о замыкании полны предрассудков и суеверий как загадочный мир, стоящий особняком внутри JavaScript, который могут познать только самые храбрые души. Но на самом деле — это всего лишь стандартный и почти очевидный факт о том, как писать

код в среде лексической области видимости, где функции являются значениями и могут свободно передаваться куда угодно.

**Замыкание — это когда функция может запомнить и иметь доступ к своей лексической области видимости даже тогда, когда она вызывается вне своей лексической области видимости.**

Замыкания могут сбить нас с толку, например в циклах, если мы не озабочимся тем, чтобы распознавать их и то как они работают. Но они еще и являются весьма мощным инструментом, дающим доступ к шаблонам, таким как *модули*в их различных формах.

Модули требуют две ключевых характеристики: 1) внешнюю функцию-обертку, которую будут вызывать, чтобы создать закрытую область видимости 2) возвращаемое значение функции-обертки должно включать в себя ссылку на не менее чем одну внутреннюю функцию, у которой потом будет замыкание на внутреннюю область видимости обертки.

Теперь вы сможете заметить замыкания повсюду в своем существующем коде и у нас теперь есть возможность обнаруживать и использовать все их преимущества!

# Приложение А: Динамическая область видимости

В главе 2 мы говорили о "динамической области видимости" как о противопоставлении модели "лексической области видимости", которая и есть то, как работает область видимости в JavaScript (а по факту и во многих других языках).

Мы кратко рассмотрим динамическую область видимости, чтобы крепко усвоить разницу. Но, важнее то, что динамическая область видимости — ближайшая родственница другого механизма (`this`) в JavaScript, который мы рассмотрели в книге "*this и прототипы объектов*".

Как мы уже отметили в главе 2, лексическая область видимости — это набор правил о том, как именно *Движок* может искать переменную и как он ее может найти. Ключевая характеристика лексической области видимости — она определяется на этапе написания кода (предполагая, что вы не жульничаете с помощью `eval()` или `with`).

Динамическая область видимости похоже означает, и не зря, что есть модель, при помощи которой область видимости можно определить динамически во время выполнения, вместо статического определения при написании кода. То, что нам нужно. Давайте отразим это в коде:

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

function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

Лексическая область видимости хранит информацию о том, что RHS-ссылка на `a` в `foo()` будет разрешена в глобальную переменную `a`, что приведет к тому, что будет выведено значение 2.

Динамическая область видимости, напротив, не интересуется тем как и где были объявлены функции и области видимости, а скорее интересуется тем **откуда они буду вызываться**. Иными словами, здесь цепочка областей видимости основана на стеке вызовов, а не на вложенности областей видимости в коде.

Итак, если бы в JavaScript была динамическая область видимости, то когда выполнилась бы `foo()`, теоретически приведенный код привел бы к выводу 3.

```
function foo() {
    console.log( a ); // 3 (not 2!)
```

```
}
```

```
function bar() {
    var a = 3;
    foo();
}

var a = 2;

bar();
```

Как такое может быть? А всё потому, что когда `foo()` не может разрешить ссылку на переменную `a`, вместо поднятия по цепочке вложенных лексических областей видимости, она взирается вверх по стеку вызовов, чтобы найти откуда `foo()` была вызвана. Поскольку `foo()` вызывалась из `bar()`, она проверяет переменные в области видимости `bar()` и находит там `a` со значением 3.

Странно? Возможно вы так и подумали, на какой-то момент.

Но это всего лишь потому, что вы возможно работали только с (или по меньшей мере глубоко изучили) кодом, который работает в лексической области видимости. Поэтому то динамическая область видимости и кажется чужой. Если бы вы писали код на языке с динамической областью видимости, для вас всё это показалось бы естественным, а лексическая область видимости показалось бы чудаковатой.

Чтобы внести ясность, в JavaScript нет, на самом деле, динамической области видимости. В нем есть лексическая область видимости. Проще некуда. Но механизм работы `this` немного похож на динамическую область видимости.

Ключевое сравнение: **лексическая область видимости определяется временем написания кода, тогда как динамическая область видимости (и `this!`) определяется во время выполнения**. Лексическую область видимости интересует где функция была объявлена, а динамическую — откуда была вызвана функция.

И наконец: `this` интересует как была вызвана функция, что показывает как близко связаны механизм `this` с идеей динамической области видимости. Чтобы изучить во всех подробностях `this`, прочтите книгу "*this и прототипы объектов*".

# Приложение В: Полифилинг блочной области видимости

В главе 3 мы исследовали блочную область видимости. Мы отметили, что операторы `with` и `catch` оба являются крошечными примерами блочной области видимости, которые существуют в JavaScript с тех пор как появился ES3.

Но в ES6 был представлен `let`, который окончательно дал полные, неограниченные возможности применять блочную область видимости в нашем коде. Есть много впечатляющих вещей, как функциональных, так и стилистических, которые появились благодаря блочной области видимости.

Но что если мы хотим использовать блочную область видимости в пред-ES6 окружении?

Представим такой код:

```
{  
    let a = 2;  
    console.log( a ); // 2  
}  
  
console.log( a ); // ReferenceError
```

Он отлично работает в ES6 окружении. Но можем ли мы также сделать в пред-ES6? `catch` — вот ответ.

```
try{throw 2}catch(a){  
    console.log( a ); // 2  
}  
  
console.log( a ); // ReferenceError
```

Ого! Какой это уродливый, странновыглядящий код. Тут `try/catch`, которые используются, чтобы принудительно вызвать ошибку, но эта "ошибка" — всего лишь значение 2, а затем объявление переменной, которая получает это значение в блоке `catch(a)`. Мозг: взорван!

Все правильно, в блоке `catch` есть блочная область видимости, которая похоже может использоваться как полифилинг для блочной области видимости в пред-ES6 окружении.

"Но...", - скажете вы. "...никто не хочет писать ужасный код подобный этому!" Это правда. Никто не пишет такой код, который выдает компилятор CoffeeScript. Но суть не в этом.

Суть в том, что утилиты могут транспилировать код на ES6, чтобы он мог работать в пред-ES6 окружении. Можно писать код с блочными областями видимости и извлекать преимущества из такой функциональности и дать возможность утилитам во время сборки проекта

позаботиться о том, чтобы сгенерировать код, который действительно будет *работать* после публикации.

Это и в самом деле предпочтительный путь миграции для всего (кхм, большей части) ES6: использовать транспилятор кода чтобы брать код на ES6 и выдавать ES5-совместимый код на период перехода от пред-ES6 к ES6.

## Traceur

---

Гугл поддерживает проект, называемый "Traceur", единственной задачей которого является транспиляция возможностей ES6 в пред-ES6 (в основном ES5, но не только!) для повседневного использования. Комитет TC39 полагается на этот инструмент (и другие), чтобы проверять на практике семантику тех возможностей, которые он выпускает.

Во что же превратит Traceur наш код? Вы угадали!

```
{
  try {
    throw undefined;
  } catch (a) {
    a = 2;
    console.log( a );
  }
}

console.log( a );
```

Так что с использованием таких утилит мы можем получать все преимущества блочной области видимости независимо от того, будет ли это работать только в ES6 или нет, потому что `try/catch` используется (и работает именно так) со временем ES3.

## Блоки: неявные против явных

---

В главе 3 мы обозначили некоторые потенциальные проблемы с обслуживаемостью/рефакторингом когда представили блочную область видимости. А есть ли другой путь получить преимущества блочной области видимости, но уменьшив эти недостатки?

Рассмотрим еще одну альтернативную форму `let`, называемую "let-блок" или "оператор `let`" (в противоположность "объявлению `let`", рассмотренным ранее).

```
let (a = 2) {
  console.log( a ); // 2
}

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

Вместо неявного "угона" существующего блока `let`-оператор создает явный блок со своей собственной областью видимости. Но явный блок выделяется не только этим и возможно

более удобным рефакторингом кода, с ним код получается чище грамматически, с помощью принудительного переноса всех определений наверх блока. Это облегчает понимание любого блока, а также того, что попадает в область его видимости, а что — нет.

Как шаблон, он отражает подход, когда многие люди используют область видимости функции и они вручную перемещают/поднимают все свои объявления var вверх функции. let-оператор помещает их в начало блока намеренно и если вы не используете объявления let, разбросанные повсюду как попало, то объявления в блочной области видимости немного легче находить и управлять ими.

Но есть проблема! let в форме оператора не включен в ES6. И официальный компилятор Traceur также не принимает такую форму кода как корректную.

У нас есть два варианта. Можно отформатировать код используя ES6-совместимый синтаксис и добавить немного дисциплины в коде:

```
/*let*/ { let a = 2;
          console.log( a );
}

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

Но инструменты призваны решать наши проблемы. Поэтому вторым вариантом будет писать явно блоки оператора let и позволить утилите сконвертировать их в корректный, работающий код.

Поэтому, я создал утилиту, названную "let-er" для решения этой единственной проблемы. let-er — транспилятор кода на этапе сборки, но его единственной задачей является находить let-операторы и транспилировать их. Она оставит в целости и сохранности весь остальной ваш код, включая любые let-объявления. Вы можете безопасно пользоваться let-er как первым звеном транспилиляции ES6, а затем передать код во что-то подобное Traceur если надо.

Более того, в let-er есть опция настройки --es6, при включении которой (по умолчанию выключена), меняется получаемый код. Вместо полифильного хака try/catch из ES3, let-er возьмет наш код и выдаст полностью ES6-совместимый, без всяких хаков:

```
{
  let a = 2;
  console.log( a );
}

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

Так что вы можете начать пользоваться let-er прямо сейчас и выпускать код под все пред-ES6 среды, а когда вам требуется только ES6, можно добавить опцию и сразу же получать только ES6-код.

И что более важно, **вы можете использовать более предпочтительную и более явную форму let-оператора** даже не смотря на то, что он не является официальной частью какой-либо версии ES (пока что).

## › Производительность

---

Позвольте мне напоследок добавить пару слов о производительности `try/catch` и/или чтобы рассмотреть вопрос: "почему бы просто не использовать IIFE для создания области видимости?"

Во-первых, производительность `try/catch` *ниже*, но нет ни одного разумного предположения, что в этом случае так и есть, или даже что *так будет всегда* в таких случаях. Поскольку официальный подтвержденный TC39 ES6-транспилятор использует `try/catch`, команда Traceur попросила Chrome улучшить производительность `try/catch` и у них конечно же есть мотивация так и сделать.

Во-вторых, IIFE — не справедливое равноценное сравнение с `try/catch`, поскольку функция, обернутая вокруг любого обычного кода, меняет значение внутри этого кода у операторов `this`, `return`, `break` и `continue`. IIFE — не замена в повседневных задачах. Ее можно использовать вручную только в особых случаях.

В итоге, вопрос превращается в такой: нужна ли вам блочная область видимости или нет. Если нужна, эти утилиты дадут вам такую возможность. Если нет, продолжайте использовать `var` и кодировать!

[Google Traceur](#)

[let-er](#)

# Приложение С: Лексический this

---

Хотя эта книга и не рассматривает механизм `this` во всех деталях, есть одна тема в ES6, которая связывает `this` с лексической областью видимости существенным образом, по которому мы быстро пробежимся.

ES6 добавляет особую синтаксическую форму объявления функции, названную "стрелочная функция". Она выглядит примерно так:

```
var foo = a => {
    console.log( a );
};

foo( 2 ); // 2
```

Так называемая "жирная стрелка" часто упоминается как сокращение для утомительно длинного (сарказм) ключевого слова `function`.

Но есть кое-что более важное в стрелочных функциях, что не имеет ничего общего с экономией символов в вашем коде.

В двух словах, этот код страдает от одной проблемы:

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
        console.log( this.id );
    }
};

var id = "not awesome";

obj.cool(); // awesome

setTimeout( obj.cool, 100 ); // not awesome
```

Проблема заключается в потере привязки `this` в функции `cool()`. Есть разные пути решения этой проблемы, но наиболее частое решение — `var self = this;`.

Выглядит это примерно так:

```
var obj = {
    count: 0,
    cool: function coolFn() {
        var self = this;

        if (self.count < 1) {
            setTimeout( function timer(){
```

```

        self.count++;
        console.log( "красиво?" );
    }, 100 );
}
};

obj.cool(); // красиво?

```

Чтобы не слишком углубляться в подробности, "решение "var self = this всего лишь избавляется целиком от всей проблемы понимания и правильного использования привязки `this`, а взамен возвращается к чему-то более удобному для нас: лексической области видимости. `self` становится всего лишь идентификатором, который может быть определен с помощью лексической области видимости и замыкания, и не заботится о том, что случится с привязкой `this` по пути.

Люди не любят писать подробно особенно то, что делают снова и снова. Таким образом, мотивацией ES6 является помочь в облегчении таких сценариев и разумеется в *устранении* общих проблем идиом, таких как эта.

Решение в ES6, стрелочная функция, вводит поведение, называемое "лексический `this`".

```

var obj = {
  count: 0,
  cool: function coolFn() {
    if (this.count < 1) {
      setTimeout( () => { // стрелочная функция, выигрышный вариант?
        this.count++;
        console.log( "красиво?" );
      }, 100 );
    }
};
obj.cool(); // красиво?

```

Вкратце, стрелочные функции ведут себя совсем не как обычные функции в том, что касается их привязки к `this`. Они отбрасывают все обычные правила для привязки `this`, а взамен берут значение `this` из их непосредственной окружающей области видимости, неважно из какой.

Так что в этом примере кода стрелочная функция не получает свой `this` непривязанным каким-то непредсказуемым путем, она всего лишь "наследует" привязку `this` функции `cool()` (что правильно, если мы вызываем ее так, как показано выше!).

Несмотря на то, что это делает код короче, моя точка зрения в том, что стрелочные функции — всего лишь на самом деле закодированная в синтаксис языка распространенная ошибка разработчиков, которая приводит к тому, чтобы запутать и соединить правила "привязки `this`" с правилами "лексической области видимости".

Другими словами: зачем искать неприятности и ударяться в словоблудие используя парадигму кодирования стиля `this`, всего лишь чтобы подрезать ему крылья смешивая его с

лексическими ссылками. Кажется естественным принять тот или иной подход для любой конкретной части кода, а не смешивать их в одном и том же месте.

**Примечание:** еще один недостаток стрелочных функций в том, что они анонимны, не именованы. Загляните в главу 3, чтобы ознакомиться с причинами почему анонимные функции менее предпочтительны, чем именованные.

Более подходящий подход, с моей точки зрения, к этой "проблеме", использовать и рассматривать механизм `this` правильно.

```
var obj = {
  count: 0,
  cool: function coolFn() {
    if (this.count < 1) {
      setTimeout( function timer(){
        this.count++; // `this` безопасен из-за `bind(..)`
        console.log( "еще красивее" );
      }.bind( this ), 100 ); // смотри, `bind()`!
    }
  }
};

obj.cool(); // еще красивее
```

Чтобы вы ни предпочли: новое поведение лексического `this` стрелочных функций или испытанный и верный `bind()`, важно отметить, что стрелочные функции — **не только** сокращение написания "function".

У них есть *намеренная разница в поведении*, которую необходимо изучить и понимать, и если мы их выбираем, то использовать по максимуму их возможности.

Теперь, когда мы полностью понимаем образование лексической области видимости (и замыкания!), понять лексический `this` будет проще простого!

# Приложение D: Благодарности

Есть множество людей, которых нужно поблагодарить за то, что появилась на свет эта книга и вся серия.

Во-первых, я должен поблагодарить мою жену Кристен Симпсон (Christen Simpson) и двух моих детей Итана (Ethan) и Эмили (Emily), за то, что мирились с тем, что их папу вечно надо было отрывать от компьютера. Даже когда я не писал книги, моя одержимость JavaScript приклеивала мой взгляд к экрану больше, чем следовало. То время, которое я занял у моей семьи, и есть причина, по которой эти книги могут так глубоко и полностью объяснить JavaScript для вас, читатель. Я в большом долгу перед своей семьей.

Хочу поблагодарить моих редакторов в O'Reilly, а именно Simon St.Laurent и Brian MacDonald, как и остальных в команде редакторов и маркетинга. Работать с ними — одно удовольствие, и хочется особенно поблагодарить их за создание подходящих условий во время этого эксперимента с написанием "open source"-книги, за редактирование и выпуск.

Спасибо всем тем, кто участвовал в улучшении этой серии книг присылая предложения по редактированию и корректировке, включая Shelley Powers, Tim Ferro, Evan Borden, Forrest L. Norvell, Jennifer Davis, Jesse Harlin и многих других. Большое спасибо Shane Hudson за написание предисловия к этой серии книг.

Спасибо бесчисленным участникам сообщества, включая членов комитета TC39, кто поделился столько многим с нами и особенно за терпеливое отношение к моим бесконечным вопросам и изысканиям. John-David Dalton, Jurij "kangax" Zaytsev, Mathias Bynens, Axel Rauschmayer, Nicholas Zakas, Angus Croll, Reginald Braithwaite, Dave Herman, Brendan Eich, Allen Wirfs-Brock, Bradley Meck, Domenic Denicola, David Walsh, Tim Disney, Peter van der Zee, Andrea Giammarchi, Kit Cambridge, Eric Elliott и многие другие, я упомянул лишь малую часть.

Серия книг *Вы не знаете JS* родилась на Kickstarter, поэтому я также хочу поблагодарить всех моих (почти) 500 щедрых инвесторов, без которых эта серия книг не появилась бы:

Jan Szpila, nokiko, Murali Krishnamoorthy, Ryan Joy, Craig Patchett, pdqtrader, Dale Fukami, ray hatfield, R0drigo Perez [Mx], Dan Petitt, Jack Franklin, Andrew Berry, Brian Grinstead, Rob Sutherland, Sergi Meseguer, Phillip Gourley, Mark Watson, Jeff Carouth, Alfredo Sumaran, Martin Sachse, Marcio Barrios, Dan, AimelyneM, Matt Sullivan, Delnatte Pierre-Antoine, Jake Smith, Eugen Tudorancea, Iris, David Trinh, simonstl, Ray Daly, Uros Gruber, Justin Myers, Shai Zonis, Mom & Dad, Devin Clark, Dennis Palmer, Brian Panahi Johnson, Josh Marshall, Marshall, Dennis Kerr, Matt Steele, Erik Slagter, Sacah, Justin Rainbow, Christian Nilsson, Delapouite, D.Pereira, Nicolas Hoizey, George V. Reilly, Dan Reeves, Bruno Laturner, Chad Jennings, Shane King, Jeremiah Lee Cohick, od3n, Stan Yamane, Marko Vucinic, Jim B, Stephen Collins, Ægir Þorsteinsson, Eric Pederson, Owain, Nathan Smith, Jeanetteurphy, Alexandre ELISÉ, Chris Peterson, Rik Watson, Luke Matthews, Justin Lowery, Morten Nielsen, Vernon Kesner, Chetan Shenoy, Paul Treguing, Marc Grabanski, Dion Almaer, Andrew Sullivan, Keith Elsass, Tom Burke, Brian Ashenfelter, David Stuart, Karl Swedberg, Graeme, Brandon Hays, John Christopher, Gior, manoj reddy, Chad Smith, Jared Harbour, Minoru TODA, Chris Wigley, Daniel Mee, Mike, Handyface, Alex Jahraus, Carl Furrow, Rob Foulkrod, Max Shishkin, Leigh Penny Jr., Robert

Ferguson, Mike van Hoenselaar, Hasse Schougaard, rajan venkataguru, Jeff Adams, Trae Robbins, Rolf Langenhuijzen, Jorge Antunes, Alex Koloskov, Hugh Greenish, Tim Jones, Jose Ochoa, Michael Brennan-White, Naga Harish Muvva, Barkóczy Dávid, Kitt Hodsdon, Paul McGraw, Sascha Goldhofer, Andrew Metcalf, Markus Krogh, Michael Mathews, Matt Jared, Juanfran, Georgie Kirschner, Kenny Lee, Ted Zhang, Amit Pahwa, Inbal Sinai, Dan Raine, Schabse Laks, Michael Tervoort, Alexandre Abreu, Alan Joseph Williams, NicolasD, Cindy Wong, Reg Braithwaite, LocalPCGuy, Jon Friskics, Chris Merriman, John Pena, Jacob Katz, Sue Lockwood, Magnus Johansson, Jeremy Crapsey, Grzegorz Pawłowski, nico nuzzaci, Christine Wilks, Hans Bergren, charles montgomery, Ariel בראב-לכוב Fogel, Ivan Kolev, Daniel Campos, Hugh Wood, Christian Bradford, Frédéric Harper, Ionuț Dan Popa, Jeff Trimble, Rupert Wood, Trey Carrico, Pancho Lopez, Joël kuijten, Tom A Marra, Jeff Jewiss, Jacob Rios, Paolo Di Stefano, Soledad Penades, Chris Gerber, Andrey Dolganov, Wil Moore III, Thomas Martineau, Kareem, Ben Thouret, Udi Nir, Morgan Laupies, jory carson-burson, Nathan L Smith, Eric Damon Walters, Derry Lozano-Hoyland, Geoffrey Wiseman, mkeehner, KatieK, Scott MacFarlane, Brian LaShomb, Adrien Mas, christopher ross, Ian Littman, Dan Atkinson, Elliot Jobe, Nick Dozier, Peter Wooley, John Hoover, dan, Martin A. Jackson, Héctor Fernando Hurtado, andy ennamorato, Paul Seltmann, Melissa Gore, Dave Pollard, Jack Smith, Philip Da Silva, Guy Israeli, @megalithic, Damian Crawford, Felix Gliesche, April Carter Grant, Heidi, jim tierney, Andrea Giammarchi, Nico Vignola, Don Jones, Chris Hartjes, Alex Howes, john gibbon, David J. Groom, BBox, Yu 'Dilys' Sun, Nate Steiner, Brandon Satrom, Brian Wyant, Wesley Hales, Ian Pouncey, Timothy Kevin Oxley, George Terezakis, sanjay raj, Jordan Harband, Marko McLion, Wolfgang Kaufmann, Pascal Peuckert, Dave Nugent, Markus Liebelt, Welling Guzman, Nick Cooley, Daniel Mesquita, Robert Syvarth, Chris Coyier, Rémy Bach, Adam Dougal, Alistair Duggin, David Loidolt, Ed Richer, Brian Chenault, GoldFire Studios, Carles Andrés, Carlos Cabo, Yuya Saito, roberto ricardo, Barnett Klane, Mike Moore, Kevin Marx, Justin Love, Joe Taylor, Paul Dijou, Michael Kohler, Rob Cassie, Mike Tierney, Cody Leroy Lindley, tofuji, Shimon Schwartz, Raymond, Luc De Brouwer, David Hayes, Rhys Brett-Bowen, Dmitry, Aziz Khoury, Dean, Scott Tolinski - Level Up, Clement Boirie, Djordje Lukic, Anton Kotenko, Rafael Corral, Philip Hurwitz, Jonathan Pidgeon, Jason Campbell, Joseph C., SwiftOne, Jan Hohner, Derick Bailey, getify, Daniel Cousineau, Chris Charlton, Eric Turner, David Turner, Joël Galeran, Dharma Vagabond, adam, Dirk van Bergen, dave ❤️🎶★ furf, Vedran Zakanj, Ryan McAllen, Natalie Patrice Tucker, Eric J. Bivona, Adam Spooner, Aaron Cavano, Kelly Packer, Eric J, Martin Drenovac, Emilis, Michael Pelikan, Scott F. Walter, Josh Freeman, Brandon Hudgeons, vijay chennupati, Bill Glennon, Robin R., Troy Forster, otaku\_coder, Brad, Scott, Frederick Ostrander, Adam Brill, Seb Flippence, Michael Anderson, Jacob, Adam Randlett, Standard, Joshua Clanton, Sebastian Kouba, Chris Deck, SwordFire, Hannes Papenberg, Richard Woeber, hnzz, Rob Crowther, Jedidiah Broadbent, Sergey Chernyshev, Jay-Ar Jamon, Ben Combee, luciano bonachela, Mark Tomlinson, Kit Cambridge, Michael Melgares, Jacob Adams, Adrian Bruinhout, Bev Wieber, Scott Puleo, Thomas Herzog, April Leone, Daniel Mizieliński, Kees van Ginkel, Jon Abrams, Erwin Heiser, Avi Laviad, David newell, Jean-Francois Turcot, Niko Roberts, Erik Dana, Charles Neill, Aaron Holmes, Grzegorz Ziolkowski, Nathan Youngman, Timothy, Jacob Mather, Michael Allan, Mohit Seth, Ryan Ewing, Benjamin Van Treese, Marcelo Santos, Denis Wolf, Phil Keys, Chris Yung, Timo Tijhof, Martin Lekvall, Agendine, Greg Whitworth, Helen Humphrey, Dougal Campbell, Johannes Harth, Bruno Girin, Brian Hough, Darren Newton, Craig McPheat, Olivier Tille, Dennis Roethig, Mathias Bynens, Brendan Stromberger, sundeep, John Meyer, Ron Male, John F Croston III, gigante, Carl Bergenhem, B.J. May, Rebekah Tyler, Ted Foxberry, Jordan Reese, Terry Suitor, afeliz, Tom Kiefer, Darragh Duffy, Kevin Vanderbeken, Andy Pearson, Simon Mac Donald, Abid Din, Chris Joel, Tomas Theunissen, David Dick, Paul Grock, Brandon Wood, John Weis, dgrebb, Nick Jenkins, Chuck Lane, Johnny Megahan, marzsman, Tatu Tamminen,

Geoffrey Knauth, Alexander Tarmolov, Jeremy Tymes, Chad Auld, Sean Parmelee, Rob Staenke, Dan Bender, Yannick derwa, Joshua Jones, Geert Plaisier, Tom LeZotte, Christen Simpson, Stefan Bruvik, Justin Falcone, Carlos Santana, Michael Weiss, Pablo Viloslada, Peter deHaan, Dimitris Iliopoulos, seyDoggy, Adam Jordens, Noah Kantrowitz, Amol M, Matthew Winnard, Dirk Ginader, Phinam Bui, David Rapson, Andrew Baxter, Florian Bougel, Michael George, Alban Escalier, Daniel Sellers, Sasha Rudan, John Green, Robert Kowalski, David I. Teixeira (@ditma, Charles Carpenter, Justin Yost, Sam S, Denis Ciccale, Kevin Sheurs, Yannick Croissant, Pau Fracés, Stephen McGowan, Shawn Searcy, Chris Ruppel, Kevin Lamping, Jessica Campbell, Christopher Schmitt, Sablons, Jonathan Reisdorf, Bunni Gek, Teddy Huff, Michael Mullany, Michael Fürstenberg, Carl Henderson, Rick Yoesting, Scott Nichols, Hernán Ciudad, Andrew Maier, Mike Stapp, Jesse Shawl, Sérgio Lopes, jsulak, Shawn Price, Joel Clermont, Chris Ridmann, Sean Timm, Jason Finch, Aiden Montgomery, Elijah Manor, Derek Gathright, Jesse Harlin, Dillon Curry, Courtney Myers, Diego Cadenas, Arne de Bree, João Paulo Dubas, James Taylor, Philipp Kraeutli, Mihai Păun, Sam Gharegozlu, joshjs, Matt Murchison, Eric Windham, Timo Behrmann, Andrew Hall, joshua price, Théophile Villard

Эта серия книг выпускается в стиле "open source", включая редактирование и выпуск. Мы отаем дань благодарности GitHub за предоставление такой возможности для сообщества!

Еще раз спасибо всех бесчисленным людям, которых я не перечислил по имени, но кого я тем ни менее должен поблагодарить. Пусть эта серия книг будет "принадлежать" всем нам и служить вкладом в увеличение информированности и понимания языка JavaScript, на благо все нынешних и будущих вкладчиков в общее дело сообщества.