

Оглавление

Введение	11
Задача12
Благодарности14
О книге22
Типографские соглашения23
Использование программного кода примеров24
От издательства24
ЧАСТЬ 1. ОБЛАСТЬ ВИДИМОСТИ И ЗАМЫКАНИЯ.....	25
Предисловие	26
Глава 1. Что такое область видимости?	28
Немного теории компиляторов29
Разбираемся в областях видимости31
Участники31
Туда и обратно32
Немного терминологии33
Общение Движка с Областью видимости36
Упражнение37
Вложенная область видимости38
Метафоры39
Ошибки41

Итоги.42
Ответ на упражнение.43
Глава 2. Лексическая область видимости	44
Стадия лексического анализа.45
Поиск.47
Искажение лексической области видимости48
eval.49
with52
Быстродействие.55
Итоги.56
Глава 3. Функциональные и блочные области видимости	57
Области видимости из функций57
Как скрыться у всех на виду.59
Предотвращение конфликтов.61
Функции как области видимости.64
Анонимные и именованные функциональные выражения66
Немедленный вызов функциональных выражений67
Блоки как области видимости.70
with72
try/catch73
let.74
const.80
Итоги.80
Глава 4. Поднятие.	82
Курица или яйцо?82
Компилятор наносит ответный удар.84
Сначала функции.87
Итоги.89

Глава 5. Замыкание области видимости	90
Просветление91
Технические подробности92
Теперь я вижу96
Циклы и замыкания99
Снова о блочной области видимости102
Модули103
Современные модули109
Будущие модули111
Итоги.113
Приложение А. Динамическая область видимости	115
Приложение Б. Полифилы для блочной области видимости	118
Tracur120
Неявные и явные блоки120
Быстродействие.123
Приложение В. Лексическое this	124
ЧАСТЬ 2. THIS И ПРОТОТИПЫ ОБЪЕКТОВ	129
Предисловие	130
Глава 6. Что такое this?	133
Для чего нужно this?133
Путаница135
Сама функция.136
Область видимости141
Что такое this?143
Итоги.144

Глава 7. this обретает смысл!	145
Место вызова	145
Ничего кроме правил	147
Связывание по умолчанию	147
Неявное связывание	149
Явное связывание	154
Связывание new	158
Все по порядку	161
Определение this	166
Исключения связывания	167
Игнорирование this	167
Косвенные ссылки	170
Мягкое связывание	171
Лексическое поведение this	173
Итоги	176
Глава 8. Объекты	177
Синтаксис	177
Тип	178
Встроенные объекты	179
Содержимое	182
Вычисление имен свойств	184
Свойства и методы	185
Массивы	188
Дублирование объектов	189
Дескрипторы свойств	192
Неизменяемость	197
[[Get]]	200
[[Put]].	202
Геттеры и сеттеры	203
Существование	206

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

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

5 Замыкание области видимости

Хочется верить, что вы открыли эту страницу с очень прочным, основательным пониманием того, как работают области видимости.

А теперь мы обратимся к невероятно важной, но хронически упускаемой из виду, *почти мифологической* части языка: *замыканиям* (closures). Если вы следили за нашим обсуждением лексической области видимости до настоящего момента, замыкания могут показаться вам чем-то банальным, почти разочаровывающим. За волшебным занавесом прячется человек¹, и вы определенно увидите его. Нет, его фамилия не Крокфорд²!

Но если у вас еще остались неразрешенные вопросы о лексических областях видимости, самое время вернуться и перечитать главу 2, прежде чем двигаться дальше.

¹ Отсылка к Гудвину из «Волшебника страны Оз» Фрэнка Баума. — *Примеч. пер.*

² https://ru.wikipedia.org/wiki/Крокфорд,_Дуглас. — *Примеч. пер.*

Просветление

Для читателей, которые имеют некоторый опыт работы на JavaScript, но, скорее, всего никогда не осознавали концепции замыканий в полной мере, понимание замыканий может показаться чем-то вроде nirваны, к достижению которой должен стремиться любой разработчик.

Много лет назад я неплохо освоил JavaScript, но понятия не имел о том, что такое замыкания. Вроде бы у языка существовала какая-то другая тайная сторона, которая обещала дать мне еще больше того, что у меня уже было, и эти слухи дразнили и искушали меня. Помню, как я читал исходный код ранних фреймворков, пытаюсь разобраться в том, как они на самом деле работают. Помню, как в моем мозгу впервые начало проявляться некое подобие «модульного паттерна». Эти моменты «Ах, вот оно как!» и сейчас вспоминаются достаточно ярко.

Тогда я не знал одного секрета, на осознание которого у меня ушли годы и которым я сейчас хочу поделиться с вами: *замыкания постоянно находятся рядом с вами в JavaScript, вам нужно только признать и принять их*. Замыкания — не какой-то новый дополнительный инструмент, для которого вам придется изучать новый синтаксис и паттерны. Нет, замыкания это даже не оружие, искусство владения которым нужно будет постигать в тренировках, как Люк постигал владение Силой.

Замыкания возникают в результате написания кода, полагающегося на лексическую область видимости. Они просто возникают сами собой. Вам даже не нужно намеренно создавать замыкания, чтобы пользоваться ими. Замыкания постоянно создаются и используются за вас в вашем коде. Вам не хватает только правильного внутреннего контекста, чтобы узнавать, принимать и использовать замыкания по вашей воле.

Момент просветления должен выглядеть примерно так: «О, замыкания уже встречаются сплошь и рядом в моем коде. Наконец-то я их вижу». Понимание замыканий немного напоминает то, как Нео впервые видит матрицу.

Технические подробности

Но довольно гипербол и отсылка к фильмам.

Ниже приводится определение того, что необходимо знать для того, чтобы понимать замыкания и узнавать их в программах.

Замыкание — способность функции запоминать свою лексическую область видимости и обращаться к ней даже тогда, когда функция выполняется вне своей лексической области видимости.

Несколько примеров помогут проиллюстрировать это определение.

```
function foo() {
    var a = 2;

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

    bar();
}

foo();
```

После наших обсуждений вложенных областей видимости этот код должен казаться знакомым. Функция `bar()` обладает доступом к переменной `a` во внешней области видимости из-за правил поиска лексической области видимости (в данном случае это поиск RHS-ссылки).

Это и есть замыкание?

Чисто с технической точки зрения... *возможно*. Но если вспомнить наше приведенное выше определение «того, что вам нужно знать»... не совсем. Я думаю, что обращение `bar()` к `a` лучше всего объясняется правилами поиска лексической области видимости, а эти правила являются лишь *составной частью* (хотя и важной) того, что называется замыканием.

С чисто теоретической точки зрения о приведенном фрагменте можно сказать, что функция `bar()` обладает *замыканием* над областью видимости `foo()` (а на самом деле и над остальными областями видимости, доступными для нее, например, глобальной областью видимости в данном случае). Если взглянуть на ситуацию несколько иначе, можно сказать, что `bar()` обладает замыканием над областью видимости `foo()`. Почему? Потому что функция `bar()` вложена в `foo()`. Просто и понятно.

Однако замыкания, определяемые таким образом, не видны напрямую, и мы не видим использования замыкания в этом фрагменте. Лексическая область видимости хорошо видна, но замыкание остается загадочной неуловимой тенью где-то за кодом. Давайте рассмотрим код, который выводит замыкание на свет:

```
function foo() {
  var a = 2;

  function bar() {
    console.log( a );
  }

  return bar;
}

var baz = foo();

baz(); // 2 -- Вы только что увидели замыкание.
```

Функция `bar()` обладает доступом лексической области видимости к внутренней области видимости `foo()`. Но затем мы берем `bar()` (саму функцию) и передаем ее *как значение*. В этом случае `return` возвращает сам объект функции, на который ссылается `bar`.

После выполнения `foo()` возвращенное значение (наша внутренняя функция `bar()`) присваивается переменной с именем `baz`, после чего происходит вызов `baz()`, что, естественно, означает вызов нашей внутренней функции `bar()`, просто по другому идентификатору.

Конечно, функция `bar()` выполняется. Но в данном случае она выполняется *за пределами* объявленной лексической области видимости.

После выполнения `foo()` обычно мы ожидаем, что вся внутренняя область видимости `foo()` исчезает, потому что мы знаем, что движок применяет уборщик мусора, который освобождает неиспользуемую память. Так как содержимое `foo()` на первый взгляд не используется, кажется естественным, что оно должно считаться утраченным.

Но «волшебство» замыканий не позволяет этому случиться. Внутренняя область видимости на самом деле *продолжает* использоваться, и поэтому не пропадает. Кто использует ее? Сама функция `bar()`.

Благодаря тому, где она была объявлена, функция `bar()` обладает замыканием лексической области видимости над внутренней областью видимости `foo()`. Поэтому данная область видимости продолжает существовать для `bar()`, что позволяет обратиться к ней в любой последующий момент времени.

`bar()` все еще содержит ссылку на эту область видимости, и эта ссылка называется *замыканием*.

Через несколько микросекунд при вызове `baz` (то есть вызове внутренней функции, которая называется `bar`) эта переменная

обладает доступом к лексической области видимости, определяемой на стадии написания программы, так что она может обратиться к переменной `a`, как и следовало ожидать.

Функция вызывается за пределами своей лексической области видимости. Замыкание позволяет функции продолжить обращаться к лексической области видимости, определенной на стадии написания программы.

Конечно, все разнообразные способы передачи функций как значений и их вызова в других точках программы являются примерами проявления/использования замыканий.

```
function foo() {
  var a = 2;

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

  bar( baz );
}

function bar(fn) {
  fn(); // смотрите, замыкание!
}
```

Мы передаем внутреннюю функцию `baz` функции `bar`, а потом вызываем эту внутреннюю функцию (теперь она называется `fn`), после чего для наблюдения ее замыкания над внутренней областью видимости `foo()` обращаемся к `a`.

Все эти передачи функций также могут быть косвенными.

```
var fn;

function foo() {
  var a = 2;
```

```
function baz() {
    console.log( a );
}

fn = baz; // baz присваивается глобальной переменной
}

function bar() {
    fn(); // смотрите, замыкание!
}

foo();

bar(); // 2
```

Какой бы механизм ни использовался для *транспортировки* внутренней функции за пределы ее области видимости, она поддерживает ссылку на область видимости, в которой была изначально объявлена, — и при каждом ее выполнении будет задействована эта ссылка.

Теперь я вижу

Предыдущие фрагменты кода выглядят искусственно сконструированными для демонстрации *использования замыканий*. Но я обещал вам нечто большее, чем новую эффектную игрушку. Я говорил, что замыкания постоянно окружают вас в существующем коде. А теперь *убедимся* в этом.

```
function wait(message) {

    setTimeout( function timer(){
        console.log( message );
    }, 1000 );

}

wait( "Hello, closure!" );
```

Мы берем внутреннюю функцию (с именем `timer`) и передаем ее `setTimeout(..)`. Однако функция `timer` имеет замыкание над областью видимости `wait(..)`, вследствие чего эта функция подерживает и использует ссылку на переменную `message`.

Через тысячу миллисекунд после того, как функция `wait` была выполнена, а ее внутренняя область видимости должна была давно исчезнуть, анонимная функция все еще имеет область замыкания над этой областью видимости.

Где-то глубоко во внутренней реализации движка встроенная функция `setTimeout(..)` содержит ссылку на параметр — возможно, с именем `fn`, или `func`, или что-нибудь в этом роде. Движок переходит к вызову этой функции, которая вызывает нашу внутреннюю функцию `timer`, а ссылка на лексическую область видимости все еще остается.

Замыкание.

Или если вы относитесь к числу сторонников jQuery (или любого другого фреймворка JS, если на то пошло):

```
function setupBot(name,selector) {
    $( selector ).click( function activator(){
        console.log( "Activating: " + name );
    } );
}
```

```
setupBot( "Closure Bot 1", "#bot_1" );
setupBot( "Closure Bot 2", "#bot_2" );
```

Не знаю, какой код пишете вы, но мне регулярно приходится писать код такого рода, так что ситуация абсолютно реалистична!

В любое время и в любом месте, где вы интерпретируете функции (которые обращаются к своим соответствующим лексическим областям видимости) как полноправные значения и передаете их,

скорее всего, вы видите, как эти функции используют замыкания. Таймеры, обработчики событий, запросы Ajax, средства межоконной передачи сообщений, веб-работники и любые другие асинхронные (или синхронные) задачи — каждый раз, когда вы передаете *функцию обратного вызова*, будьте готовы к тому, что к ней прилагается замыкание!



В главе 3 был описан паттерн IIFE. Хотя часто говорят, что паттерн IIFE (сам по себе) является проявлением замыкания, я бы частично не согласился, по нашему предыдущему определению.

```
var a = 2;

(function IIFE(){
    console.log( a );
})();
```

Этот код работает, но он не является проявлением замыкания. Почему? Потому, что функция (которой здесь присвоено имя IIFE) не выполняется за пределами своей лексической области видимости. Она вызывается прямо в той области действия, в которой была объявлена (внешняя/глобальная область видимости также содержит *a*). Переменная *a* находится посредством обычного поиска по лексической области видимости, а не с использованием замыкания.

Хотя формально замыкания происходят во время объявления, они остаются незаметными для наблюдателя, словно дерево, падающее в лесу, когда этого никто не видит и не слышит.

Хотя выражения IIFE *сами по себе* не являются проявлениями замыканий, они безусловно создают области видимости и остаются одним из самых популярных инструментов для создания областей видимости, для которых могут создаваться замыкания.

Таким образом, выражения IIFE тесно связаны с замыканиями даже при том, что они не являются проявлениями замыканий.

Отложите книгу, дорогой читатель. У меня есть для вас задание. Откройте какой-нибудь код JavaScript, написанный вами в последнее время. Взгляните на свои функции, передаваемые как значения; определите, где вы уже использовали замыкания (возможно, даже не сознавая этого).

Я подожду.

Теперь вы увидели!

Циклы и замыкания

Самый частый и канонический пример, используемый для демонстрации замыканий, основан на обычном цикле `for`.

```
for (var i=1; i<=5; i++) {  
    setTimeout( function timer(){  
        console.log( i );  
    }, i*1000 );  
}
```



Статические анализаторы кода часто жалуются при размещении функций в циклах, потому что многие разработчики допускают ошибки, связанные с непониманием замыканий. Я объясню, как это правильно делать, в полной мере используя силу замыканий. Но этот нюанс часто теряется для статических анализаторов кода, которые предполагают, что вы сами не понимаете, что делаете.

По духу этого фрагмента кода можно было бы ожидать, что он выведет числа 1, 2, ..., 5 — по одному каждую секунду.