КАЙЛ
СИМПСОН

# ОБЛАСТЬ ВИДИМОСТИ И ЗАМЫКАНИЯ

2-Е ИЗДАНИЕ

{ ВЫ ПОКА ЕЩЕ НЕ ЗНАЕТЕ JS }

# You Don't Know JS Yet: Scope & Closures

Get to know JS

Kyle Simpson

КАЙЛ
СИМПСОН

# ОБЛАСТЬ ВИДИМОСТИ И ЗАМЫКАНИЯ

2-Е МЕЖДУНАРОДНОЕ ИЗДАНИЕ

{ ВЫ ПОКА ЕЩЕ НЕ ЗНАЕТЕ JS }

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

*Кайл Симпсон*

# {Вы пока еще не знаете JS}
# Область видимости и замыкания. 2-е межд. изд.

*Перевел с английского Е.Матвеев*


| | |
|---|---|
| Заведующая редакцией | *Ю. Сергиенко* |
| Ведущий редактор | *А. Юринова* |
| Корректоры | *М. Одинокова, Г. Шкатова* |
| Верстка | *Е. Неволайнен* |

ББК 32.988.02-018
УДК 004.738.5

**Симпсон Кайл**

С37 {Вы пока еще не знаете JS} Область видимости и замыкания. 2-е межд. издание. — СПб.: Питер, 2022. — 240 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1876-2



Вы пока еще не знаете JS. И Кайл Симпсон признается, что тоже его не знает (по крайней мере полностью)… И никто не знает. Но все мы можем начать работать над тем, чтобы узнать его лучше. Сколько бы времени вы ни провели за изучением языка, всегда можно найти что-то еще, что стоит изучить и понять на другом уровне. Вы уже прочитали «Познакомьтесь, JavaScript»? Тогда откройте вторую книгу серии «Вы пока еще не знаете JS», чтобы познакомиться поближе с первым из трех столпов JavaScript — системой областей видимости и функциональными замыканиями, а также с мощным паттерном проектирования «Модуль». Пора освоить правила лексических областей видимости для размещения переменных и функций в правильных позициях. И заглянуть на более низкий уровень, ведь магия с хранением состояния модулей базируется на замыканиях, использующих систему лексических областей видимости.



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

ISBN 978-1647862213 англ.
ISBN 978-5-4461-1876-2





Изготовлено в России. Изготовитель: ООО «Прогресс книга». Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург, Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373. Дата изготовления: 08.2021. Наименование: книжная продукция. Срок годности: не ограничен. Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные. Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01. Подписано в печать 13.08.21. Формат 60x90/16. Бумага офсетная. Усл. п. л. 15,000. Тираж 1000. Заказ

# Оглавление

# Благодарности

Прежде всего спасибо моей жене и детям. Их постоянная поддержка позволила мне продолжать работу. Также хочу поблагодарить 500 бэкеров первого издания «Вы не знаете JS» (YDKJS) на Kickstarter, а также сотни тысяч людей, которые купили и прочли эти книги. Без вашей финансовой поддержки второе издание не состоялось бы. Также спасибо интервьюеру из одной соцсети с птичьим названием, который сказал, что я «недостаточно знаю JS», чем помог мне выбрать название для серии книг.

Своей карьерой я в значительной мере обязан Марку Грабански (Marc Grabanski) и FrontendMasters. Много лет назад Марк оказал мне доверие и помог сделать первые шаги в области преподавания. Если бы не он, я не начал бы писать книги! Frontend Masters является главным спонсором «Вы все еще не знаете JS») (2-е издание). Спасибо вам, Frontend Masters (и Марк!).

Наконец, мой редактор Саймон Сен-Лоран (Simon St.Laurent) помог мне определиться с первоначальным замыслом серии YDKJS и стал редактором моей первой книги. Поддержка и советы Саймона оказали на меня серьезное влияние, и именно благодаря им я в значительной мере сформировался как автор. Прошло много лет с тех пор, как за выпивкой в Driskill родился замысел YDKJS. Спасибо тебе, Саймон, за все эти годы, что ты указывал мне путь и улучшал эти книги!

# Предисловие

Когда я смотрю на книги на полке, сразу вижу любимые. Любимые книги всегда потертые. Переплет надорван, на замусоленных страницах — пятна от пролитых напитков. Удивительно, что самые любимые книги выглядят так, словно о них меньше всего заботятся, хотя, честно говоря, все совсем наоборот.

Первое издание этой книги — одно из самых моих любимых. Она невелика, но переплет уже начал разваливаться. Страницы потрепаны, уголки загибаются. Это явно не книга на один раз. Я снова и снова возвращалась к ней в течение многих лет, прошедших с момента ее издания.

Для меня она также стала вехой моего личного прогресса в изучении JavaScript. Впервые она попалась мне в руки в 2014 году; на тот момент я была знакома с основными концепциями, но, откровенно говоря, глубина моего понимания не могла сравниться с тем, что описано в этой тоненькой книжице.

Шли годы. И хотя мне порой казалось, будто мои профессиональные навыки вовсе не улучшаются, мне все же удалось разобраться со всеми концепциями из книги. Я улыбаюсь, сознавая, какой путь прошла под этим руководством. Стало очевидно, что между моей любовью к этой книге и моим бережным отношением к ней была обратно пропорциональная связь.

Когда Кайл предложил написать вступление ко 2-му изданию, я была ошеломлена. Нечасто вам предлагают написать что-то о книге, которая оказала такое влияние на ваше собственное понимание и карьеру. Помню тот день, когда впервые поняла суть

замыканий; первый раз, когда я успешно воспользовалась ими. Тогда я была горда собой, отчасти из-за того, что меня привлекала симметрия этой идеи. Я была восхищена замыканиями еще до того, как взялась за эту книгу. Но просто написать рабочий код — совсем не то же самое, что глубоко изучить концепции. Эта книга улучшила мое понимание фундаментальных вещей и помогла их мастерски освоить.

Книга получилась обманчиво короткой. То, что она настолько мала, — весьма удобно, так как материал очень информационно насыщен. Рекомендую побольше времени выделить на усвоение каждой страницы. Не торопитесь. Относитесь к книге со всем вниманием — чтобы она стала такой же потрепанной и зачитанной, как и моя.

*Сара Дрейснер (Sarah Drasner), руководитель группы DX, Netlify*

# Вступление

Вашему вниманию предлагается 2-е издание снискавшей популярность серии книг «Вы не знаете JS»: «Вы пока еще не знаете JS» (YDKJSY).

Если вы уже читали предыдущее издание, то заметите, что в этом появился обновленный подход к изложению с подробными описаниями того, что изменилось в JS за последние 5 лет.

Я надеюсь и верю, что вы все еще сохраняете стремление изучить JS и разобраться в том, как же он устроен.

Если вы читаете эти книги впервые, я рад, что они попались вам на глаза. Подготовьтесь к увлекательному путешествию по закоулкам JavaScript.

Если вы недавно занимаетесь программированием или JS, то учтите, что эти книги не задумывались как «деликатный вводный курс по JavaScript». Временами материал становится сложным и требующим серьезных усилий, и многие темы рассматриваются намного глубже, чем в книгах для новичков. Книга может пригодиться всем читателям независимо от уровня подготовки, но я писал ее с прицелом на то, что вы уже знакомы с JS, а ваш практический опыт работы с этим языком составляет хотя бы полгода, если не больше.

## Части языка

В этих книгах я намеренно отошел от традиционного подхода, в котором рассматриваются *хорошие части* языка. Нет, это не

означает, что мы будем рассматривать только *плохие части* — скорее рассматриваться будут **все части**.

Возможно, вы слышали (или сами считаете), что JS — глубоко ущербный язык, плохо спроектированный и непоследовательно реализованный. Многие считают, что это худший из популярных языков; что никто не пишет код JS добровольно, а только из-за того, что он занял свое место в сети. Это смехотворные, нездоровые и высокомерные утверждения.

Миллионы разработчиков ежедневно пишут код JavaScript, и многие из них уважают и ценят этот язык.

Как и у любого великого языка, у него есть как выдающиеся достоинства, так и недостатки. Даже сам создатель JavaScript Брендан Эйх сожалеет по поводу некоторых частей и называет их ошибками. Но он заблуждается: они вовсе не были ошибками. В наши дни JS стал тем, чем он стал — самым распространенным, а следовательно, самым влиятельным языком программирования, — именно из-за *всех этих частей*.

Не ведитесь на утверждения, будто вам следует изучить и использовать только небольшой набор *хороших частей*, а от всего плохого нужно держаться подальше. Не ведитесь на шарлатанство «X — это новый Y», будто с появлением в языке некоторой новой возможности все предшествующее использование старой функциональности мгновенно устаревает и отмирает. Не слушайте, когда кто-то вам говорит, что ваш код «не современен», потому что в нем еще не используется функция стадии 0, предложенная лишь несколько недель назад!

Все части JS полезны. Некоторые части полезнее других. Некоторые требуют действовать более внимательно и осознанно.

На мой взгляд, абсурдно даже пытаться стать по-настоящему эффективным разработчиком JavaScript, используя только узкий срез возможностей языка. Можно ли представить рабочего с полным ящиком инструментов, который пользуется только молотком,

а отвертку и рулетку презирает, считая их недостойными? Это просто глупо.

Я утверждаю, что изучать нужно все части JavaScript и пользоваться ими там, где они уместны! И я даже наберусь смелости предложить: выбросьте все книги, в которых говорится обратное.

## Название?

Какой же смысл заложен в название серии?

Я не пытаюсь обидеть вас, ставя под сомнение ваш уровень знания или понимания JavaScript. Я не предполагаю, что вы не можете или не сможете изучить JavaScript. Я не хвастаюсь некими секретными тайными знаниями, которыми обладаю только я и еще несколько избранных.

Серьезно, все это реальные реакции на название оригинальной серии, которые появились еще до того, как книги увидели свет. И они совершенно необоснованны.

Главный смысл названия «Вы пока еще не знаете JS» — подчеркнуть, что большинство разработчиков JS не тратит время на то, чтобы по-настоящему понять, как работает написанный ими код. Они знают, что код *работает* — он выдает желаемый результат. Но они либо не понимают, *как* он работает, либо, что еще хуже, руководствуются неточной ментальной моделью, которая дает сбой при ближайшем рассмотрении.

Я предлагаю вам спокойно, но вдумчиво отложить все свои допущения по поводу JS, взглянуть на язык свежим взглядом и подойти к нему с заново пробужденной любознательностью. Спрашивайте себя «почему?» каждый раз, когда пишете строчку. Почему она работает именно так, а не иначе? Почему один способ лучше или уместнее пяти-шести других возможных решений? Почему все «лидеры мнений» предлагают делать X в вашем коде, но выясняется, что вариант Y оказывается лучше?

Я добавил в название «пока» не только потому, что это второе издание, но и из-за того, что в конечном итоге я хочу, чтобы книги вселяли в вас надежду, а не убивали ее.

Не думаю, что JS вообще возможно знать полностью. Это не достижение, которое необходимо получить, а цель, к которой нужно стремиться. Не думайте, что вы все узнаете о JS и на этом все закончится; нет, вы просто продолжаете учиться, все чаще практикуясь в написании кода. И чем глубже вы погружаетесь, тем чаще возвращаетесь к тому, что изучали ранее, и переосмысливаете его с позиций более опытного разработчика.

Рекомендую сформировать особую систему взглядов на JavaScript (и на разработку в целом): вы никогда не освоите его полностью, но можете (и должны) работать над тем, чтобы приблизиться к этой цели. Этот путь растянется на всю вашу карьеру разработчика и даже дальше.

Вы всегда можете знать JS лучше, чем сейчас. Надеюсь, именно эту мысль передают книги серии YDKJSY.

## Миссия

На самом деле не нужно обосновывать, почему разработчики должны относиться к JS серьезно — думаю, язык уже доказал, что заслуживает статуса первоклассного среди языков программирования.

Важно обосновать другое, более глобальное утверждение, и эти книги пытаются справиться с этой задачей.

Я обучал более 5000 разработчиков из групп и компаний по всему миру более чем в 25 странах на шести континентах. Мне часто приходилось видеть, что главным фактором считается только результат программы, а не то, как программа написана или как/почему она работает.

Мой опыт не только как разработчика, но и как преподавателя говорит мне: вы всегда можете повысить эффективность своего

труда, если четко будете понимать, как работает ваш код (а не просто добиваться того, чтобы он выдавал желаемый результат).

Иначе говоря, «*код достаточно хорош, чтобы работать*» — не то же самое, что «*код достаточно хорош*» (и не должно быть тем же самым).

Всем разработчикам постоянно приходится мучиться с каким-нибудь блоком кода, который по неизвестной причине работает неправильно. Но слишком часто разработчики JS обвиняют язык, вместо того чтобы винить себя за нехватку понимания. Эти книги служат вопросом и ответом: почему произошло именно *это* и как нужно действовать, чтобы произошло *вот это*.

Моя миссия — дать возможность каждому разработчику JS полностью контролировать написанный им код, понять его и программировать сознательно и ясно.

## Путь

Некоторые из вас начали читать эту книгу с целью изучить все шесть книг от начала и до конца.

Давайте немного скорректируем этот план. Последовательное чтение книг серии не входило в мои намерения. Материал в них освоить не так-то просто, потому что JavaScript — язык мощный, замысловатый и порой достаточно сложный. Никому не удастся *загрузить* всю эту информацию в мозг за один проход, вы неизбежно забудете почти все прочитанное. Лучше даже не пытаться.

Мой совет: не торопитесь. Возьмите одну главу, прочитайте ее полностью от начала до конца, потом вернитесь и перечитайте раздел за разделом. Разберите код и идеи в каждом разделе. Если вы столкнетесь с чем-то сложным, лучше провести несколько дней за усвоением, повторным чтением и тренировками, а потом продолжить изучение.

На каждую главу можно выделить неделю или две, на каждую книгу — месяц или два, на всю серию — год и более, и даже в этом случае вы еще не выжмете из YDKJSY все возможное.

Не читайте эти книги взахлеб; будьте терпеливы. Чередуйте чтение с практикой: применяйте знания в рабочих задачах или собственных проектах. Оспаривайте мои идеи, возражайте, а самое главное — не соглашайтесь со мной! Организуйте учебную группу или клуб. Проводите мини-семинары в своем офисе. Пишите посты о прочитанном. Обсудите эти темы на локальных встречах JS.

Моя цель не навязать вам свое мнение. Скорее я хочу выработать у вас собственное мнение и умение его отстаивать. Вы не сможете достичь *этой цели* скоростным чтением. На это уйдет немало времени. Вы будете двигаться вперед шаг за шагом, пока изучаете, размышляете и возвращаетесь к прочитанному. Эти книги были задуманы как путеводитель по JavaScript от вашего текущего местонахождения в знаниях о языке до точки более глубокого понимания. А теперь самая интересная часть: чем глубже вы понимаете JS, тем больше вопросов у вас появится и тем больше придется изучать!

Я очень рад, что вы отправляетесь в путешествие, и для меня большая честь, что вы сочли мои книги достойными своего внимания и решили довериться им. Пришло время начать *изучение* JS!

## От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

Мы снабдили некоторые изображения QR-кодами, чтобы вы могли посмотреть их цветные версии.

# 1 Что такое область видимости?

Вероятно, к тому моменту, когда вы уже написали несколько первых программ, то уже освоились с созданием переменных и хранением в них значений. Работа с переменными — один из фундаментальных навыков в программировании!

Но возможно, вы не уделяли особого внимания механизмам, которые используются движком для организации и управления этими переменными. Я говорю не о выделении памяти компьютером, а совсем о другом: как JS узнает, какие переменные доступны для любой конкретной команды, и как она поступает, обнаруживая две переменные с одинаковыми именами?

Ответы на подобные вопросы воплощаются в системе четко определенных правил, называемых *областью видимости* (scope). В этой книге будут подробно рассмотрены все аспекты области видимости — как она работает, для чего она нужна, каких скрытых в ней ловушек стоит остерегаться. После этого будут представлены распространенные паттерны области видимости, определяющие структуру программ.

А начнем мы с описания того, как движок JS обрабатывает наши программы перед запуском.

## О книге

Добро пожаловать во вторую книгу серии «Вы пока еще не знаете JS»! Если вы уже прочитали первую книгу «Познакомьтесь, JavaScript», вы на верном пути! Если нет, рекомендую начать с первой книги, которая подготовит вас к этому материалу.

Книга посвящена первому из трех столпов языка JS: системе областей видимости и ее функциональным замыканиям, а также мощному паттерну проектирования «Модуль».

JS обычно относят к категории интерпретируемых языков сценариев, поэтому предполагается, что большинство программ JS обрабатывается за один проход «сверху вниз». Но в действительности JS разбирается/компилируется в отдельной фазе до начала выполнения. Решения автора кода в отношении того, как размещать переменные, функции и блоки относительно друг друга, анализируются с учетом правил области видимости в исходной фазе разбора/компиляции. Полученная структура кода обычно не зависит от условий стадии выполнения.

Функции JS сами по себе являются полноправными значениями; их можно присваивать и передавать точно так же, как числа или строки. Но так как эти функции содержат переменные и обращаются к ним, они поддерживают свою исходную область видимости независимо от того, в какой точке программы эти функции будут выполняться в конечном итоге. Эта концепция называется замыканием.

Модули представляют паттерн организации кода, для которого характерны открытые методы с привилегированным доступом (через замыкание) к скрытым переменным и функциям во внутренней области видимости модуля.

## Компилируемые и интерпретируемые языки

Конечно, вы уже слыхали о *компиляции кода*. Но скорее всего, процесс компиляции кажется вам чем-то вроде «черного ящика», в который с одного конца подается исходный код, а с другого — выскакивают исполняемые программы.

Но ничего загадочного или волшебного здесь нет. Компиляция кода — последовательность шагов, которая обрабатывает исходный код и преобразует его в набор инструкций, понятных компьютеру. Как правило, весь исходный код преобразуется одновременно, и эти инструкции сохраняются как результат (обычно в файле), который может быть выполнен позднее.

Возможно, вы также слышали, что код может интерпретироваться. Чем же *интерпретация* отличается от *компиляции*?

Интерпретация решает примерно ту же задачу, что и компиляция, — в том смысле, что преобразует вашу программу в набор инструкций, понятных машине. При этом используется другая модель обработки. Если при компиляции обрабатывается сразу вся программа, при интерпретации исходный код преобразуется строка за строкой; после выполнения каждой строки или команды происходит немедленный переход к следующей строке исходного кода.

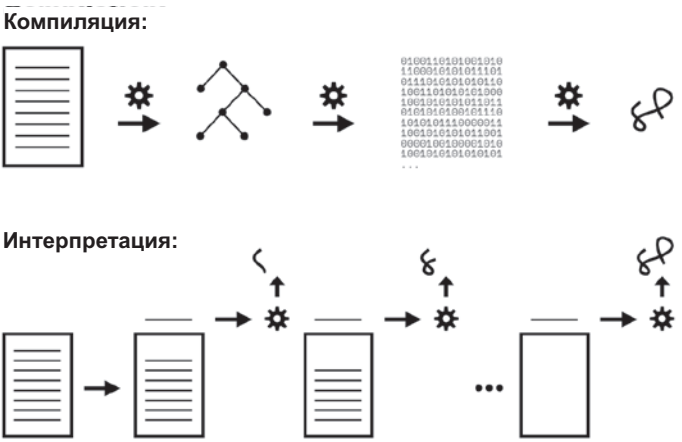


**Рис. 1.** Компилируемый и интерпретируемый код

На рис. 1 представлена сравнительная схема интерпретации и компиляции программ.

Являются ли эти две модели обработки взаимоисключающими? В общем случае да. Но проблема скрывает ряд нюансов, потому

что интерпретация в действительности может принимать другие формы помимо простой построчной обработки исходного кода. Современные движки JS в действительности применяют различные сочетания компиляции и интерпретации при обработке программ JS.

Напомню, что эта тема рассматривалась в главе 1 книги «Познакомьтесь, JavaScript». Там был сделан вывод, что JS точнее всего описывать как компилируемый язык. Для удобства читателей в следующих разделах мы вернемся к этому утверждению и расширим его.

## Компиляция кода

Но для начала необходимо понять, зачем мы вообще говорим о том, компилируется JS или нет?

Видимость в основном определяется в фазе компиляции, поэтому понимание связи между компиляцией и выполнением играет ключевую роль для понимания областей видимости.

В классической теории компиляторов обработка программы компилятором состоит из трех основных этапов:

1. **Разбиение на лексемы/лексический разбор:** строка символов разбивается на осмысленные (для языка) фрагменты, называемые лексемами. Для примера возьмем программу: `var a = 2;`. Скорее всего, эта программа будет разбита на следующие лексемы: `var`, `a`, `=`, `2` и `;`. Пробелы могут сохраняться как лексемы, а могут и не сохраняться — это зависит от того, содержательны они или нет.

   Различия между разбиением на лексемы (tokenizing) и лексическим разбором (lexing) достаточно тонкие и академические, но это зависит от того, как идентифицируются лексемы — с учетом состояния или без. Проще говоря, если для определения того, должен ли символ `a` считаться отдельной лексемой или просто частью другой лексемы, подсистема разбиения на лексемы должна активизировать правила разбора с учетом состояния, это будет лексический разбор.

2. **Разбор:** поток (массив) лексем преобразуется в дерево вложенных элементов, которые совместно представляют грамматическую структуру программы. Оно называется *абстрактным синтаксическим деревом* (AST).

   Например, дерево для программы `var a = 2;` может начинаться с узла верхнего уровня с именем `VariableDeclaration`, имеющего дочерний узел с именем `Identifier` (со значением `a`) и еще один дочерний узел `AssignmentExpression`, у которого есть дочерний узел `NumericLiteral` (со значением `2`).

3. **Генерирование кода:** AST преобразуется в исполняемый код. Эта часть очень сильно изменяется в зависимости от языка, целевой платформы и других факторов. Движок JS получает только что описанное дерево AST для `var a = 2;` и преобразует его в набор машинных команд для фактического *создания* переменной с именем `a` (включая резервирование памяти и т. д.) и последующего сохранения значения в `a`.

Подробности реализации движка JS (использование ресурсов системной памяти и т. д.) намного глубже, чем мы здесь рассматриваем. Основное внимание будет уделяться наблюдаемому поведению наших программ, а управление более глубокими абстракциями системного уровня будет доверено движку JS.

Принципы работы движка JS намного сложнее, они далеко не сводятся *только* к этим трем стадиям. В процессе разбора и генерирования кода выполняются особые действия для оптимизации быстродействия (например, исключение избыточных элементов). Более того, код даже может быть перекомпилирован и заново оптимизирован в ходе выполнения.

В общем, я привожу только общую картину. Но вскоре вы увидите, почему те подробности, которые здесь *рассматриваются* (даже на высоком уровне), важны для изучаемой темы.

Движку JS недоступна такая роскошь, как лишнее время для выполнения их работы и оптимизаций, потому что компиляция JS

не выполняется на отдельной стадии до выполнения, как в других языках. Обычно она должна выполняться за считаные микросекунды (или менее!) прямо перед выполнением кода. Чтобы обеспечить наилучшее быстродействие в этих условиях, движок JS использует всевозможные трюки (например, метод JIT с отложенной компиляцией и даже горячей перекомпиляцией); все они выходят за рамки нашего изложения.

### Две фазы

В самой простой формулировке самое важное замечание, которое можно сделать по поводу обработки программ JS, заключается в том, что она выполняется (как минимум) в две фазы: сначала разбор/компиляция, затем выполнение.

Отделение фазы разбора/компиляции от последующей фазы выполнения — наблюдаемый факт, а не какая-то теория или субъективное мнение. Хотя спецификация JS не требует компиляции явно, она требует поведения, которое по сути реально только при подходе «компиляция с последующим выполнением».

Чтобы убедиться в этом, можно понаблюдать за тремя характеристиками программ: синтаксические ошибки, ранние ошибки и поднятие (hoisting).

### Синтаксические ошибки

Рассмотрим следующую программу:

```
var greeting = "Hello";

console.log(greeting);

greeting = ."Hi";
// SyntaxError: unexpected token .
```

Программа ничего не выводит (сообщение `"Hello"` не выводится), а вместо этого выдает ошибку `SyntaxError` о неожиданной лексеме `.`

прямо перед строкой `"Hi"`. Так как синтаксическая ошибка происходит после правильно сформированной команды `console.log(..)`, если бы код JS выполнялся при построчном выполнении программы сверху вниз, можно было бы ожидать, что сообщение `"Hello"` будет выведено перед выдачей синтаксической ошибки. Но этого не происходит.

Движок JS может узнать о синтаксической ошибке в третьей строке, перед выполнением первой и второй строк, только в одном случае: если движок JS сначала разбирает всю программу до того, как будет выполнена любая из ее частей.

## Ранние ошибки

Теперь следующая программа:

```
console.log("Howdy");

saySomething("Hello","Hi");
// Неперехваченная ошибка SyntaxError: одинаковые имена
// параметров недопустимы в этом контексте

function saySomething(greeting,greeting) {
    "use strict";
    console.log(greeting);
}
```

Сообщение `"Howdy"` не выводится, несмотря на правильно сформированную команду.

Вместо этого, как и во фрагменте из предыдущего раздела, перед выполнением программы выдается ошибка `SyntaxError`. В данном случае это объясняется тем, что строгий режим (включенный только для функции `saySomething(..)`) запрещает среди прочего функции с одинаковыми именами параметров; в нестрогом режиме это всегда было разрешено.

Выданная ошибка не является синтаксической ошибкой, обусловленной неправильно сформированной последовательностью лексем (как `."Hi"` выше), но в строгом режиме спецификация требует выдавать раннюю ошибку до начала выполнения.

Но как движок JS узнает, что параметр `greeting` повторяется? Откуда он знает, что функция `saySomething(..)` выполняется в строгом режиме во время обработки списка параметров (директива `"use strict"` появляется позже в теле функции)?

И снова возможно только одно разумное объяснение: код полностью разбирается до начала выполнения.

## Поднятие

Наконец, рассмотрим пример:

```
function saySomething() {
    var greeting = "Hello";
    {
        greeting = "Howdy"; // здесь происходит ошибка
        let greeting = "Hi";
        console.log(greeting);
    }
}

saySomething();
// ReferenceError: невозможно обратиться к 'greeting'
// до инициализации
```

Источником ошибки `ReferenceError` является строка с командой `greeting = "Howdy"`. Дело в том, что переменная `greeting` из этой команды относится к объявлению в следующей строке `let greeting = "Hi"`, а не к предыдущей команде `var greeting = "Hello"`.

В строке, в которой выдается ошибка, движок JS может только в одном случае узнать о том, что *следующая команда* объявляет одноименную переменную (`greeting`) с блоковой видимостью: если движок JS уже обработал этот код на более раннем проходе и уже сформировал все области видимости и их связи с переменными. Такая обработка областей видимости и объявлений может быть достигнута только при обработке программы перед выполнением.

С технической точки зрения ошибка `ReferenceError` происходит из-за того, что команда `greeting = "Howdy"` обращается к пере-

менной `greeting` **слишком рано** — этот конфликт обозначается сокращением *TDZ* (Temporal Dead Zone). В главе 5 эта тема рассматривается более подробно.

Часто встречаются утверждения о том, что объявления `let` и `const` не поднимаются, как показывает только что приведенное объяснение поведения TDZ. Тем не менее это неточное утверждение. Мы еще вернемся к темам поднятия и TDZ для конструкций `let/const` в главе 5.

Надеюсь, я убедил вас в том, что программы JS разбираются до начала какого-либо выполнения. Но доказывает ли это, что они компилируются?

И это интересный вопрос. Может ли JS разобрать программу, но затем выполнить ее с *интерпретацией* операций, представленных в AST, без ее предварительной компиляции? Да, это *возможно*. Но в высшей степени маловероятно, прежде всего потому, что это будет крайне неэффективно с точки зрения быстродействия.

Трудно представить себе, что движок JS коммерческого уровня пойдет на все хлопоты с разбором программы в AST, но без последующего преобразования (компиляции) полученного дерева AST в самое эффективное (двоичное) представление для его последующего выполнения движком.

Многие придираются к этой терминологии, так как в этой области встречается множество нюансов и замечаний «хотя вообще-то...». Однако по духу и на практике то, что делает движок при обработке программ JS, имеет с компиляцией **гораздо больше общего**, чем с чем-либо другим.

Классификация JS как компилируемого языка не имеет отношения к модели распространения в двоичном исполняемом представлении (или в байт-коде). Скорее она всего лишь четко выделяет в нашем мысленном представлении фазу, в которой обрабатывается и анализируется код JS; эта фаза наблюдаемо и бесспорно выполняется до начала выполнения кода.

Если мы хотим понимать и эффективно использовать JS и области видимости, понадобятся подходящие ментальные модели того, как движок JS обходится с нашим кодом.

## С точки зрения компилятора

Зная о двухфазной обработке программы JS (сначала компиляция, потом выполнение), обратимся к тому, как движок JS идентифицирует переменные и определяет области видимости в программе в процессе компиляции.

Сначала рассмотрим простую программу JS, которая будет использоваться для анализа в нескольких ближайших главах:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

Кроме объявлений, все вхождения переменных/идентификаторов в программе играют одну из двух ролей: они являются либо *приемником* присваивания, либо *источником* значения.

Когда я только начал изучать теорию компиляторов для получения диплома computer science, в ней для этих ролей использо-

вались термины LHS (приемник) и RHS (источник) соответственно. Эти сокращения происходят от Left-Hand Side и Right-Hand Side — для левой и правой сторон оператора присваивания `=`. Однако приемники и источники присваивания не всегда буквально располагаются слева и справа от `=`, поэтому я предпочитаю термины «*приемник/источник*» вместо «левый/правый».

Как узнать, является ли переменная *приемником*? Проверьте, присваивается ли ей значение; если присваивается, то это *приемник*. Если нет, то переменная является *источником*.

Чтобы движок JS правильно обрабатывал переменные в программе, он должен сначала пометить каждое вхождение переменной как *приемник* или как *источник*. Сейчас мы разберемся, как определяется каждая из этих ролей.

## Приемники

Что делает переменную *приемником*? Пример:

```
students = [ // ..
```

Эта команда очевидно является операцией присваивания; помните, что часть `var students` полностью обрабатывается как объявление во время компиляции, а следовательно, не актуальна во время выполнения; мы опустили ее для ясности. То же относится к команде `nextStudent = getStudentName(73)`.

Но есть еще три менее очевидные операции, использующие *приемник* присваивания. Одна из них:

```
for (let student of students) {
```

Эта команда присваивает значение `student` при каждой итерации цикла. Другая ссылка на *приемник*:

```
getStudentName(73)
```

Но каким образом это является присваиванием? Присмотритесь внимательнее: аргумент `73` присваивается параметру `studentID`.

И в нашей программе осталась последняя (неочевидная) ссылка на приемник.

А вы сможете ее найти?

..
..
..

Ну как, догадались?

```
function getStudentName(studentID) {
```

Объявление функции является специальным случаем ссылки на приемник. Ее можно рассматривать как нечто похожее на `var getStudentName = function(studentID)`, но это не совсем точно.

Идентификатор `getStudentName` объявляется во время компиляции, но часть `= function(studentID)` также обрабатывается в процессе компиляции; связь между `getStudentName` и функцией автоматически создается в начале области видимости, вместо того чтобы ожидать выполнения команды присваивания `=`.

Автоматическое связывание функции с переменной называется поднятием функции (function hoisting). Эта тема подробно рассматривается в главе 5.

## Источники

Итак, мы выявили все пять ссылок на приемник в программе. Тогда все остальные ссылки на переменные должны быть ссылками на источники (потому что других вариантов нет).

Мы выяснили, что в `for (let student of students)` переменная `student` является ссылкой на приемник, а `students` — ссылкой на источник. В команде `if (student.id == studentID)` и `student`, и `studentID` являются ссылками на источник. `student` также является ссылкой на источник в команде `return student.name`.

В `getStudentName(73)` переменная `getStudentName` является ссылкой на источник (которая, как мы надеемся, будет преобразована в зна-

чение ссылки на функцию). В команде `console.log(nextStudent)` переменная `console` является ссылкой на источник, как и `nextStudent`.

Если вас интересует, `id`, `name` и `log` — свойства, а не ссылки на переменные.

В чем практический смысл разделения приемников и источников? В главе 2 мы вернемся к этой теме и посмотрим, как роль переменной влияет на последствия ее поиска (а именно если поиск завершается неудачей).

## Изменение области видимости во время выполнения

К настоящему моменту должно быть уже ясно, что область видимости определяется при компиляции программы и обычно не должна зависеть от условий на стадии выполнения. Тем не менее в нестрогом режиме технически это правило можно обойти двумя способами, изменяя области видимости программы во время выполнения.

Ни один из этих приемов *не следует* применять на практике — оба слишком опасны и слишком запутанны, и вам все равно стоит использовать строгий режим (в котором они запрещены). Но важно знать о них на тот случай, если вы столкнетесь с ними в каких-нибудь программах.

Функция `eval(..)` получает строку с кодом, который должен быть откомпилирован и выполнен «на ходу», во время выполнения программы. Если эта строка кода содержит `var` или объявление функции, эти объявления изменят текущую область видимости, в которой в настоящее время выполняется `eval(..)`:

```
function badIdea() {
    eval("var oops = 'Ugh!';");
    console.log(oops);
}
badIdea(); // Ugh!
```

Если бы вызов `eval(..)` отсутствовал, то переменная `oops` в `console.log(oops)` не существовала бы и программа выдала бы ошибку `ReferenceError`. Однако `eval(..)` изменяет область видимости функции `badIdea()` во время выполнения. Это нежелательно по многим причинам, включая снижение быстродействия из-за модификации уже откомпилированной и оптимизированной области видимости при каждом запуске `badIdea()`.

Второй трюк основан на использовании ключевого слова `with`, которое, по сути, динамически преобразует объект в локальную области видимости — его свойства интерпретируются как идентификаторы в блоке новой области видимости:

```
var badIdea = { oops: "Ugh!" };

with (badIdea) {
    console.log(oops); // Ugh!
}
```

Глобальная область видимости здесь не изменяется, но объект `badIdea` приводится в область видимости во время выполнения, а не во время компиляции, а его свойство `oops` превращается в переменную в этой области видимости. Еще раз подчеркну: это очень плохая идея по соображениям быстродействия и удобочитаемости.

Всеми силами избегайте `eval(..)` (по крайней мере, `eval(..)` с созданием объявлений) и `with`. Напомню, что ни один из этих трюков не доступен в строгом режиме, и если вы просто используете строгий режим (а это стоит делать!), искушение пропадет само собой.

## Лексические области видимости

Мы показали, что область видимости JS определяется на стадии компиляции; такая разновидность областей видимости называется лексической областью видимости. Определение «лексический» относится к фазе лексического разбора процесса компиляции, как обсуждалось ранее в этой главе.

Чтобы свести эту главу к одному полезному выводу, ключевая идея лексической области видимости заключается в том, что она

полностью определяется размещением функций, блоков и объявлений переменных относительно друг друга.

Если объявление переменной размещается внутри функции, компилятор обрабатывает его в процессе разбора функции и связывает это объявление с областью видимости функции. Если переменная объявлена с блоковой областью видимости (`let`/`const`), то она связывается с ближайшим вмещающим блоком `{..}` вместо вмещающей функции (как с ключевым словом `var`).

Более того, ссылка на переменную (в роли *приемника* или *источника*) должна разрешаться как поступившая из одной из областей видимости, которая доступна ей на *лексическом уровне*; в противном случае говорят, что переменная не объявлена (что обычно приводит к ошибке!). Если переменная не объявлена в текущей области видимости, то проверяется следующая внешняя/вмещающая область видимости. Процесс перехода на один уровень области видимости продолжается до тех пор, пока не будет найдено подходящее объявление переменной или не будет достигнута глобальная область видимости, и дальше идти уже некуда.

Важно заметить, что при компиляции не выполняются никакие *реальные* операции, связанные с резервированием памяти для областей видимости и переменных. Ни одна команда в программе еще не выполнена.

Вместо этого процесс компиляции строит план всех лексических областей видимости. Эта структура данных определяет, что потребуется программе во время ее выполнения. Можно рассматривать этот план как кодовую вставку, используемую во время выполнения, в которой определяются все области видимости (лексическое окружение) и регистрируются все идентификаторы (переменные) для каждой области видимости.

Иначе говоря, хотя области видимости определяются во время компиляции, их фактическое создание откладывается до стадии выполнения. В следующей главе будут кратко описаны концептуальные основания лексической видимости.

# 2 Лексическая видимость

Из главы 1 вы узнали, как область видимости определяется во время компиляции кода, в модели, которая называется лексической видимостью. Термин «лексическая» относится к первой стадии компиляции (лексический разбор).

Чтобы правильно *рассуждать* о программах, важно иметь прочное концептуальное основание того, как работают области видимости. Догадки и интуиция порой могут привести к правильному ответу, но в большинстве случаев результаты окажутся далеки от истины. Это явно не путь к успеху.

Как и на школьных уроках математики, одного правильного ответа недостаточно — нужно показать правильные шаги, которые привели вас к ответу! Необходимо построить точные и полезные ментальные модели, которые будут лежать в основе ваших рассуждений.

В этой главе работа *областей видимости* будет продемонстрирована с помощью нескольких метафор. Наша цель — научиться *представлять* обработку нашей программы движком JS максимально близко к тому, как этот движок работает на самом деле.

# Камешки, банки, рамки... Ну и ну!

Одна из метафор, которые, по моему опыту, хорошо помогают понять смысл областей видимости, — цветные камешки, которые раскладываются по банкам соответствующих цветов.

Представьте, что у вас есть куча красных, синих и зеленых камешков. Вы хотите разложить все камешки по банкам: красные кладутся в красную банку, зеленые — в зеленую, а синие — в синюю. Если после сортировки вам понадобится зеленый камешек, вы уже знаете, что его нужно искать в зеленой банке.

В этой метафоре камешки представляют переменные в нашей программе. Банки соответствуют областям видимости (функциям и блокам), которым мы назначили разные цвета просто для целей обсуждения. Таким образом, цвет каждого камешка определяется *цветом* области видимости, в которой этот камешек был изначально создан.

Разметим пример программы из главы 1 цветами:

```
// Внешняя/глобальная область видимости: КРАСНЫЙ

var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    // Функциональная область видимости: СИНИЙ

    for (let student of students) {
        // Область видимости цикла: ЗЕЛЕНЫЙ

        if (student.id == studentID) {
            return student.name;
        }
    }
}
```

```
var nextStudent = getStudentName(73);
console.log(nextStudent); // Suzy
```

Мы обозначили три цвета области видимости комментариями: КРАСНЫЙ (внешняя/глобальная область видимости), СИНИЙ (область видимости функции `getStudentName(..)`) и ЗЕЛЕНЫЙ (область видимости цикла `for`). Но по листингу может быть трудно распознать границы областей видимости. Чтобы вам было проще наглядно представить области видимости, на рис. 2 они обозначены рамками:

```
var students = [                                    1
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {                 2
        if (student.id == studentID) {
            return student.name;                    3
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// "Suzy"
```

**Рис. 2.** Границы областей видимости

1. Рамка 1 (КРАСНЫЙ) охватывает глобальную область видимости, которая содержит три идентификатора/переменные:

`students` (строка 1), `getStudentName` (строка 8) и `nextStudent` (строка 16).

2. Рамка 2 (СИНИЙ) охватывает область видимости функции `getStudentName(..)` (строка 8), которая содержит всего один идентификатор/переменную: параметр `studentID` (строка 8).
3. Рамка 3 (ЗЕЛЕНЫЙ) охватывает область видимости цикла `for` (строка 9), которая содержит всего один идентификатор/переменную: `student` (строка 9).

Формально параметр `studentID` не совсем принадлежит области видимости СИНИЙ (2). Эта неоднозначность будет рассмотрена в разделе «Предполагаемые области видимости» приложения А. А пока будет достаточно отнести `studentID` к области видимости СИНИЙ (2).

Границы областей видимости определяются во время компиляции на основании того, где находятся функции/блоки видимости, как они вложены друг в друга и т. д. Каждая область видимости полностью содержится в родительской области видимости — она никогда не принадлежит двум внешним областям видимости.

Цвет каждого камешка (переменной/идентификатора) определяется цветом банки, в которой он находится (объявляется), а не цветом области видимости, из которой к нему может происходить обращение (например, `students` в строке 9 и `studentID` в строке 10).

Вспомните: в главе 1 было сказано, что `id`, `name` и `log` являются свойствами, а не переменными; иначе говоря, это не камешки, разложенные по банкам, поэтому им не назначаются цвета по правилам, описанным в книге. О том, как обрабатываются обращения к свойствам, рассказано в третьей книге серии, «Объекты и классы».

Когда движок JS обрабатывает программу (во время компиляции) и обнаруживает объявление переменной, он фактически спрашивает: «В каком *цвете* (рамке, банке) я сейчас нахожусь?» Пере-

менная обозначается тем же *цветом*, что означает, что она принадлежит этой рамке/банке.

Область видимости ЗЕЛЕНЫЙ (3) полностью вложена в область СИНИЙ (2); аналогичным образом область СИНИЙ (2) полностью вложена в область КРАСНЫЙ (1). Области видимости могут вкладываться друг в друга так, как показано, до произвольной глубины, необходимой вашей программе.

Ссылки (не объявления) на переменные/идентификаторы считаются допустимыми в том случае, если подходящее объявление существует либо в текущей области видимости, либо в любой области видимости выше текущей, но не в области видимости более низкого уровня.

Выражение в области видимости КРАСНЫЙ (1) может обращаться только к переменным из КРАСНЫЙ (1), но не СИНИЙ (2) или ЗЕЛЕНЫЙ (3). Выражение в области видимости СИНИЙ (2) может обращаться к переменным из СИНИЙ (2) или КРАСНЫЙ (1), но не ЗЕЛЕНЫЙ (3). Наконец, выражение в области видимости ЗЕЛЕНЫЙ (3) может обращаться к переменным из КРАСНЫЙ (1), СИНИЙ (2) и ЗЕЛЕНЫЙ (3).

Процесс определения этих цветов во время выполнения можно на концептуальном уровне представить себе как поиск. Так как ссылка на переменную `students` в цикле `for` в строке 9 не является объявлением, цвет у нее отсутствует. Соответственно, мы спрашиваем у текущей области видимости СИНИЙ (2), присутствует ли в ней камешек с заданным именем. Так как его нет, поиск продолжается во внешней/вмещающей области видимости: КРАСНЫЙ (1). В банке КРАСНЫЙ (1) лежит камешек с именем `students`, так что ссылка на переменную `students` в цикле идентифицируется как КРАСНЫЙ (1).

Команда `if (student.id == studentID)` в строке 10 аналогичным образом содержит ссылки на переменную ЗЕЛЕНЫЙ (3) с именем `student` и переменную СИНИЙ (2) с именем `studentID`.

Движок JS на самом деле не определяет эти «цвета» во время выполнения; использованный в тексте термин «поиск» — всего лишь риторический прием, который позволяет понять суть концепций. Во время компиляции большинство ссылок на переменные соответствует уже известным областям видимости, так что их «цвет» уже определен и он хранится с каждой ссылкой для предотвращения лишнего поиска во время выполнения программы. Подробнее об этом нюансе рассказано в главе 3.

Ключевые выводы из аналогии с камешками и банками (и рамками):

- Переменные объявляются в конкретных областях видимости, что можно рассматривать как цветные камешки, разложенные по банкам соответствующих цветов.
- Любая ссылка на переменную, находящаяся в области видимости, в которой она была объявлена, или в одной из областей видимости более глубокой вложенности, будет помечена как относящаяся к тому же цвету — если только промежуточная область видимости не «заместит» объявление переменной; см. раздел «Затенение» главы 3.
- Определение «цветов» областей видимости и находящихся в них переменных происходит во время компиляции. Эта информация используется для «поиска» переменных (определения цвета камешков) во время выполнения кода.

## Дружеское общение

Другая полезная метафора для процесса анализа переменных и областей видимости, из которых они происходят, — «диалоги», происходящие внутри движка в ходе обработки и последующего выполнения кода. Мы можем «прислушаться» к этим диалогам, чтобы лучше понять на концептуальном уровне, как работают области видимости.

Послушаем, что говорят участники процесса в ходе обработки нашей программы:

- *Движок*: отвечает за компиляцию и выполнение ваших программ JavaScript.
- *Компилятор*: один из друзей *Движка*; выполняет всю тяжелую работу по разбору и генерированию кода (см. предыдущий раздел).
- *Менеджер области видимости*: еще один друг *Движка*; собирает и ведет список поиска всех объявленных переменных/идентификаторов и поддерживает набор правил, определяющих их доступность для текущего выполняемого кода.

Чтобы вы в полной мере поняли, как работает JavaScript, необходимо начать думать так, как думает Движок (и его друзья), задавать те вопросы, которые задают они, и давать такие же ответы на вопросы. Чтобы проанализировать это общение, вернемся к нашему постоянному примеру:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// Suzy
```

Посмотрим, как JS будет обрабатывать эту программу, начиная с первой команды. Массив и его содержимое — обычные литералы-значения JS (а следовательно, проблемы области видимости их вообще не касаются), поэтому нас здесь будет интересовать

только объявление `var students = [ .. ]` и инициализация/присваивание.

Обычно мы рассматриваем эту команду как единое целое, но с точки зрения *Движка* это не так. Для JS это две разные операции: одна выполняется *Компилятором* во время компиляции, а другая выполняется *Движком* во время выполнения.

Обработка этой программы *Компилятором* начинается с лексического разбора и разделения ее на лексемы, которые затем преобразуются в дерево (AST).

После того как *Компилятор* доберется до генерирования кода, ему приходится учитывать многие неочевидные подробности. Разумно предположить, что *Компилятор* сгенерирует для первой команды код, который означает примерно следующее: выделить память для переменной, связать ее с именем `students` и сохранить в этой переменной ссылку на массив. Однако это не все.

Ниже перечислены операции, которые будут выполнены *Компилятором* при обработке этой команды:

1. Встречая в программе конструкцию `var students`, *Компилятор* приказывает *Менеджеру области видимости* проверить, существует ли в этой конкретной области видимости переменная с именем `students`. Если она существует, то *Компилятор* игнорирует это объявление и двигается дальше. В противном случае *Компилятор* генерирует код, который (во время выполнения) прикажет *Менеджеру области видимости* создать новую переменную с именем `students` в этой области видимости.
2. Затем *Компилятор* генерирует код, который будет позднее выполнен *Движком*, для обработки присваивания `students = []`. Код, выполняемый *Движком*, сначала прикажет *Менеджеру области видимости* проверить, доступна ли в текущей области видимости переменная с именем `students`. Если она недоступна, *Движок* продолжает поиск в других местах (см. ниже раздел «Вложенные области видимости»). Когда *Движок* найдет переменную, оно присваивает ей ссылку на массив `[ .. ]`.

В форме диалога между *Компилятором* и *Движком* первая фаза компиляции программы выглядит примерно так:

**Компилятор:** Эй, *Менеджер области видимости* (для глобальной области видимости), я нашел формальное объявление идентификатора с именем `students`. Когда-нибудь слышал о таком?

**Менеджер области видимости (глобальный):** Нет, впервые слышу. Вот создаю специально для тебя.

**Компилятор:** Эй, *Менеджер области видимости*, я нашел формальное объявление идентификатора с именем `getStudentName`. Когда-нибудь слышал о таком?

**Менеджер области видимости (глобальный):** Тоже нет, но создам для тебя.

**Компилятор:** Эй, *Менеджер области видимости*, `getStudentName` указывает на функцию, так что нам понадобится новая область видимости.

**Менеджер области видимости (для функции):** Понял, получай новую область видимости.

**Компилятор:** Эй, *Менеджер области видимости* (для функции), я нашел объявление формального параметра для `studentID`. Слышал о таком?

**Менеджер области видимости (для функции):** Нет, но создал его в этой области видимости.

**Компилятор**: Эй, *Менеджер области видимости* (для функции), я нашел цикл `for`, которому нужна своя область видимости.

...

Диалог состоит из вопросов и ответов. *Компилятор* спрашивает текущий *Менеджер области видимости*, встречалось ли ранее объявление обнаруженного им идентификатора. Если оно не встречалось, то *Менеджер области видимости* создает эту переменную в своей области видимости. Если встречалось, то оно

фактически игнорируется, потому что *Менеджеру области видимости* делать ничего не нужно.

*Компилятор* также сигнализирует, когда он переходит между областями видимости функций или блоков, чтобы были созданы новые экземпляры области видимости и *Менеджера области видимости*.

Позднее, когда начнется выполнение программы, начнется диалог *Движка* с *Менеджером области видимости*, и этот диалог может выглядеть примерно так:

**Движок:** Эй, *Менеджер области видимости* (глобальный), ты можешь найти идентификатор `getStudentName`, чтобы я мог присвоить ему эту функцию?

**Менеджер области видимости (глобальный):** Да, вот переменная.

**Движок:** Эй, *Менеджер области видимости*, я нашел ссылку-приемник для `students`, ты когда-нибудь слышал о таком?

**Менеджер области видимости (глобальный):** Да, он был формально объявлен для этой области видимости. Вот, держи.

**Движок:** Спасибо. Я инициализирую переменную `students` значением `undefined`, так что она готова к использованию. Эй, *Менеджер области видимости* (глобальный), я нашел ссылку-приемник для переменной `nextStudent`, знаешь о ней?

**Менеджер области видимости (глобальный):** Да, она была формально объявлена для этой области видимости. Держи.

**Движок:** Спасибо. Я инициализирую переменную `nextStudents` значением `undefined`, так что она готова к использованию. Эй, *Менеджер области видимости* (глобальный), я нашел ссылку-источник для `getStudentName`, знаешь о ней?

**Менеджер области видимости (глобальный):** Да, она была формально объявлена для этой области видимости. Держи.

**Движок**: Прекрасно, значение в `getStudentName` является функцией, поэтому я ее выполню.

**Движок:** Эй, *Менеджер области видимости*, теперь нам нужно создать экземпляр области видимости функции.

...

Этот диалог — еще один обмен вопросами и ответами, в котором *Движок* сначала приказывает *Менеджеру области видимости* провести поиск поднятого (hoisted) идентификатора `getStudentName`, чтобы связать с ним функцию. Затем *Движок* продолжает спрашивать *Менеджера области видимости* о ссылке-приемнике для `students` и т. д.

Ниже приводится краткая сводка обработки команд вида `var students = [ .. ]` за два этапа:

1. *Компилятор* создает объявление переменной в области видимости (так как она еще не была ранее объявлена в текущей области видимости).
2. Во время работы *Движка* для обработки присваивания в этой команде *Движок* дает команду *Менеджеру области видимости* провести поиск переменной и инициализировать ее `undefined`, чтобы она была готова к использованию, после чего присваивает ему значение-массив.

## Вложенная область видимости

Когда наступает время выполнить функцию `getStudentName()`, *Движок* запрашивает у *Менеджера области видимости* экземпляр области видимости для этой функции, затем переходит к поиску параметра (`studentID`), чтобы присвоить ему значение аргумента 73, и т. д.

Область видимости функции `getStudentName(..)` вложена в глобальную область видимости. Блоковая область видимости для цикла `for` аналогичным образом вложена в область видимости этой функции. Области видимости могут вкладываться друг в друга на произвольную глубину так, как определяет программа.

Каждая область видимости получает собственный экземпляр *Менеджера области видимости* при каждом ее выполнении (один или несколько раз). Каждая область видимости автоматически регистрирует все свои идентификаторы в начале выполнения (это называется поднятием переменных, см. главу 5).

В начале области видимости, если какой-либо идентификатор поступил из объявления функции, эта переменная автоматически инициализируется ссылкой на ассоциированную функцию. А для любого идентификатора, поступающего из объявления `var` (в отличие от `let`/`const`), переменная автоматически инициализируется `undefined`, чтобы она могла использоваться; в противном случае переменная остается неинициализированной (т. е. в состоянии TDZ, см. главу 5) и не может использоваться до выполнения ее полного объявления и инициализации.

В команде `for (let student of students) {` идентификатор `students` является ссылкой-источником, для которого необходимо провести поиск. Но как выполнить такой поиск, ведь область видимости функции не найдет такой идентификатор?

Чтобы понять это, представим, что происходит такой разговор:

**Движок:** Эй, *Менеджер области видимости* (*для функции*), я нашел ссылку-источник для `students`, слышал о такой?

**Менеджер области видимости (для функции):** Нет, впервые слышу. Попробуй следующую внешнюю область видимости.

**Движок:** Эй, *Менеджер области видимости* (*для глобальной области видимости*), я нашел ссылку-источник для `students`, слышал о такой?

**Менеджер области видимости (для глобальной области видимости):** Да, было такое формальное объявление. Вот оно.

...

Один из ключевых аспектов лексической области видимости заключается в том, что если в любой момент ссылку на идентификатор не удается найти в текущей области видимости, происходит

обращение к следующей внешней области видимости; процесс повторяется, пока не будет обнаружен ответ или пока не будут проверены все возможные области видимости.

### Неудача при поиске

Когда *Движок* завершает перебор всех *лексически доступных* областей видимости (двигаясь наружу), но найти идентификатор так и не удается, возникает ситуация ошибки. Но в зависимости от режима программы (действует строгий режим или нет) и роли переменной (т. е. *приемник* или *источник*; см. главу 1) эта ситуация ошибки будет решена иначе.

### Путаница с неопределенностью

Если переменная является источником, безрезультатный поиск идентификатора считается необъявленной (неизвестной, отсутствующей) переменной, что всегда приводит к выдаче ошибки `ReferenceError`. Кроме того, если переменная и код являются приемником, и код выполняется в строгом режиме, переменная считается необъявленной, и в этом случае также выдается ошибка `ReferenceError`.

Сообщение об ошибке для ситуации с необъявленной переменной в большинстве сред JS выглядит так: «Reference Error: XYZ не определен». Слова «не определен» (not defined) в английском языке почти не отличаются от «неопределенный» (undefined). Тем не менее в JS это два совершенно разных понятия, и, к сожалению, это сообщение об ошибке постоянно создает путаницу.

«Не определен» в данном случае означает «не объявлен», т. е. у переменной нет подходящего формального объявления в любой *лексически доступной* области видимости. С другой стороны, «неопределенный» в действительности означает, что переменная была найдена (объявлена), но на данный момент она не содержит другого значения, поэтому по умолчанию в ней хранится значение `undefined`.

Чтобы ситуация запуталась еще сильнее, оператор `typeof` в JS возвращает строку `undefined` для ссылок на переменные в любом из этих состояний:

```
var studentName;
typeof studentName; // "undefined"

typeof doesntExist; // "undefined"
```

Эти две ссылки на переменные находятся в совершенно разных условиях, но JS безусловно мутит воду. Эта терминологическая мешанина крайне прискорбна и расстраивает. К сожалению, разработчикам JS приходится постоянно помнить об этом и понимать, с какой разновидностью «неопределенности» они имеют дело в любой момент!

## Глобальный... что?!

Если переменная является приемником, а строгий режим не действует, вступает в действие запутанное и неожиданное унаследованное поведение. К сожалению, *Менеджер области видимости* для глобальной области видимости просто создает глобальную переменную для реализации присваивания приемнику!

Пример:

```
function getStudentName() {
    // Присваивание необъявленной переменной :(
    nextStudent = "Suzy";
}

getStudentName();

console.log(nextStudent);
// "Suzy" -- увы, произвольная глобальная переменная!
```

*Диалог* в данном случае происходит так:

**Движок:** Эй, *Менеджер области видимости* (*для функции*), я нашел ссылку-приемник для `nextStudent`, знаешь про такую?

**Менеджер области видимости (для функции):** Нет, впервые слышу. Попробуй следующую внешнюю область видимости.

**Движок:** Эй, *Менеджер области видимости* (*для глобальной области видимости*), я нашел ссылку-источник для `nextStudent`, слышал о такой?

**Менеджер области видимости (для глобальной области видимости):** Нет, но поскольку мы не работаем в строгом режиме, я решил помочь и создал для тебя глобальную переменную. Вот, держи!

Кошмар.

Подобные инциденты (которые со временем почти наверняка приведут к ошибкам) — отличный пример защитных мер, предлагаемых строгим режимом, и именно из-за них не рекомендуется отказываться от использования строгого режима. В строгом режиме *Менеджер области видимости* (*для глобальной области видимости*) вместо этого должен будет ответить:

**Менеджер области видимости (для глобальной области видимости):** Нет, впервые слышу. Извини, мне придется выдать ошибку `ReferenceError`.

Присваивание необъявленной переменной является ошибкой, поэтому выдача ошибки `ReferenceError` будет правильным поведением.

Никогда не полагайтесь на произвольно создаваемые глобальные переменные. Всегда используйте строгий режим и всегда формально объявляйте ваши переменные. Тогда, если вы по ошибке попытаетесь присвоить значение необъявленной переменной, вы получите полезную ошибку `ReferenceError`.

## Развитие метафор

Чтобы наглядно представить разрешение имен с вложенными областями видимости, я использую другую метафору — здание, как на рис. 3.

Здание представляет набор вложенных областей видимости нашей программы. Первый этаж соответствует области видимости, выполняемой в настоящий момент, а верхний этаж — глобальной области видимости.

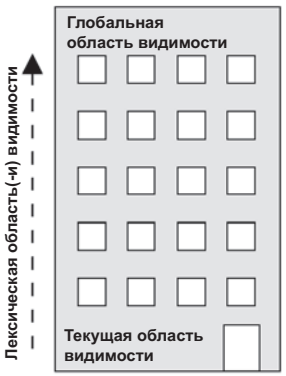


**Рис. 3.** «Здание» областей видимости

Чтобы разрешить ссылку на переменную-*источник* или переменную-*приемник*, сначала мы проверяем текущий этаж, и если переменная не найдена — поднимаемся на лифте на следующий этаж (т. е. во внешнюю область видимости), смотрим там, затем на следующий и т. д. Добравшись до последнего этажа (глобальной области видимости), вы либо находите там искомое, либо не находите. Но на этом в любом случае приходится остановиться.

## Продолжение диалога

К этому моменту вы должны уже неплохо представлять, что такое область видимости и как движок JS определяет и использует их на основании вашего кода.

Прежде чем идти дальше, обратитесь к исходному коду одного из своих проектов и воспроизведите эти диалоги. Серьезно, проговорите их вслух. Найдите друга и потренируйтесь с ним в каждой из ролей. Если кто-то из вас запутается или перестанет понимать, что происходит, проведите дополнительное время за изучением материала.

Переходя (поднимаясь) к следующей (внешней) главе, мы рассмотрим, как лексические области видимости соединяются в цепочку.

# 3 Цепочка областей видимости

В главах 1 и 2 приведено конкретное определение лексической области видимости (и ее частей), а также представлены полезные метафоры, которые помогают понять ее концептуальную основу. Прежде чем продолжать чтение этой главы, найдите кого-нибудь, кому можно объяснить (письменно или на словах), что такое лексическая область видимости и почему полезно понимать эту концепцию. Может показаться, что этот шаг можно пропустить, но мой опыт показывает, что объяснять эти идеи другим очень полезно. Это помогает нашему мозгу лучше усвоить изучаемый материал!

А теперь пришло время углубиться в технические подробности, так что должен предупредить: с этого момента описание становится намного более подробным. Но не пропускайте его, потому что оно по-настоящему дает понять, сколько вы еще не знаете об областях видимости (пока). Не торопитесь.

Чтобы напомнить контекст нашего постоянного примера, вспомним иллюстрацию вложенных областей видимости с цветными рамками из главы 2 (рис. 2).

Связи между областями видимости, вложенными между другими областями видимости, образуют *цепочку областей видимости*.

Эта цепочка определяет путь, по которому можно обращаться к переменным. Она является однонаправленной, т. е. поиск движется только снизу вверх/наружу.

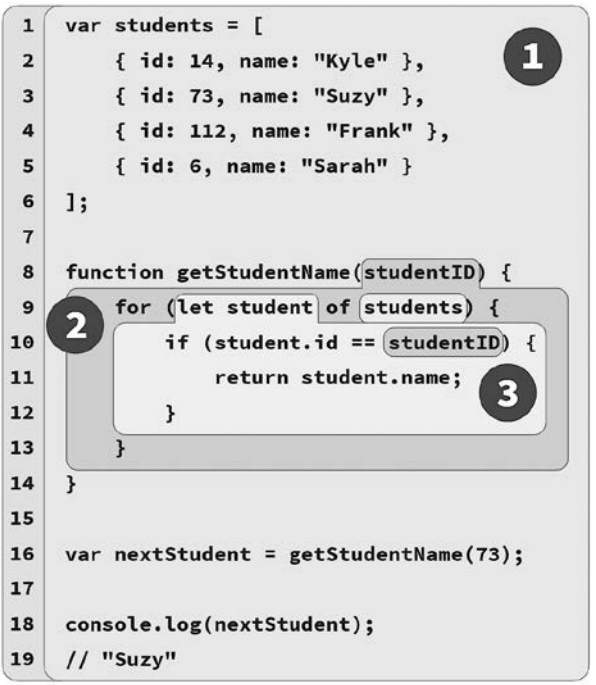


**Рис. 2 (глава 2):** Области видимости

## «Поиск» (большей частью) концептуален

На рис. 2 обратите внимание на ссылку на переменную `students` из цикла `for`. Каким образом было определено, что она относится к области видимости КРАСНЫЙ (1)?

В главе 2 процедура обращения к переменной во время выполнения была описана термином «поиск». Поиск запускался *Движком*, для чего он спрашивал у *Менеджера области видимости*

текущей области видимости, известен ли ему данный идентификатор/переменная, и продвигалось вверх/наружу по цепочке вложенных областей видимости (по направлению к глобальной области видимости), пока идентификатор/переменная не будет найден (если будет). Поиск останавливается при обнаружении первого объявления с подходящим именем в очередной области видимости.

Процесс поиска определил, что `students` принадлежит области видимости КРАСНЫЙ (1), потому что мы еще не нашли подходящее имя переменной при перемещении по цепочке областей видимости, пока не прибыли к последней глобальной области видимости КРАСНЫЙ (1).

Аналогичным образом идентификатор/переменная `studentID` в команде `if` определяется как принадлежащий области видимости СИНИЙ (2).

Такое описание процесса поиска во время выполнения хорошо работает для концептуального понимания, но на практике все обычно происходит не так.

Цвет банки, в которой лежит камешек (т. е. метаинформация о том, из какой области видимости происходит переменная), *обычно определяется* во время исходного процесса компиляции. Так как лексическая область видимости на этот момент более или менее фиксирована, цвет камешка не изменится под воздействием каких-либо факторов, возникших позднее на стадии выполнения.

Так как цвет камешка известен с момента компиляции и остается неизменным, эта информация с большой вероятностью будет храниться в узле каждой переменной в AST (или, по крайней мере, будет доступна из него); эта информация затем явно используется исполняемыми командами программы.

Иначе говоря, *Движку* (из главы 2) не нужно просматривать набор областей видимости, чтобы понять, из какой области видимости происходит переменная. Эта информация уже известна! Предот-

вращение поиска на стадии выполнения — ключевое преимущество лексических областей видимости для оптимизации быстродействия. Исполнительная система действует более эффективно, когда ей не приходится тратить время на эти поиски.

Но я совсем недавно сказал «*...обычно определяется...*», когда речь зашла об определении цвета камешка во время компиляции. В каком же случае эта информация не будет известна во время компиляции?

Возьмем ссылку на переменную, которая не объявляется ни в одной лексически доступной области видимости в текущем файле — см. книгу 1 «Познакомьтесь, JavaScript», где сказано, что каждый файл является самостоятельной программой с точки зрения компиляции JS. Если объявление не найдено, это не обязательно является ошибкой. Другой файл (программа) на стадии выполнения может объявить эту переменную в общей области видимости.

Итак, окончательное определение того, была ли переменная правильно объявлена в некоторой доступной области видимости, иногда приходится откладывать до стадии выполнения.

Любая ссылка на изначально необъявленную переменную остается в виде «неокрашенного камешка» во время компиляции файла; ее цвет невозможно определить до того, как будут откомпилированы другие файлы и не активизируется исполнительная среда приложения. Отложенный поиск в конечном итоге определит, в какой области видимости была обнаружена переменная (скорее всего, в глобальной).

Однако этот поиск потребуется выполнить не более одного раза на каждую переменную, так как никакие факторы во время выполнения не смогут изменить «цвет камешка».

В разделе «Неудача при поиске» главы 2 рассказано, что происходит, если камешек остается неокрашенным в момент выполнения ссылки на него на стадии выполнения программы.

## Затенение

В нашем постоянном примере для этих глав используются разные имена переменных для разных областей видимости. Так как имена уникальны, в каком-то смысле можно было бы с таким же успехом хранить их все в одной области видимости (например, КРАСНЫЙ (1)).

Различия между лексическими областями видимости начинают играть более важную роль, когда в программе встречаются две и более переменные с одинаковыми именами, определенные в разных областях видимости. В одной области видимости не может быть двух и более переменных с совпадающими именами; такие множественные ссылки будут интерпретироваться как одна переменная.

Таким образом, если вам потребуется создать две и более переменные с одинаковыми именами, необходимо использовать разные (часто вложенные) области видимости. И в этом случае очень важно, как эти области видимости расположены относительно друг друга.

Пример:

```
var studentName = "Suzy";

function printStudent(studentName) {
    studentName = studentName.toUpperCase();
    console.log(studentName);
}

printStudent("Frank");
// FRANK

printStudent(studentName);
// SUZY

console.log(studentName);
// Suzy
```

Переменная `studentName` в строке 1 (команда `var studentName = ..`) создает переменную в области видимости КРАСНЫЙ (1). Одноименная переменная объявляется как принадлежащая области

видимости СИНИЙ (2) в строке 3 (параметр в определении функции `printStudent(..)`).

Прежде чем читать дальше, попробуйте проанализировать этот код с применением различных приемов/метафор, представленных в книге. В частности, постарайтесь определить «цвета» (т. е. области видимости) переменных в этом фрагменте. Это полезная тренировка!

К какому цвету будет относиться `studentName` в команде присваивания `studentName = studentName.toUpperCase()` и в команде `console.log(studentName)`? Все три ссылки на `studentName` будут принадлежать СИНИЙ (2).

В концептуальном описании поиска было сказано, что он начинается с текущей области видимости, распространяется наружу/наверх и останавливается только при обнаружении подходящей переменной.

Переменная `studentName` из СИНИЙ (2) обнаруживается немедленно. Переменная `studentName` из КРАСНЫЙ (1) даже не рассматривается.

В этом проявляется ключевой аспект поведения лексических областей видимости, называемый *затенением*. Переменная (параметр) `studentName` из СИНИЙ (2) замещает переменную `studentName` из КРАСНЫЙ (1). Таким образом, параметр затеняет (затеняемую) глобальную переменную. Несколько раз повторите это предложение про себя и убедитесь в том, что вы правильно понимаете терминологию!

Вот почему повторное присваивание `studentName` влияет только на внутреннюю переменную (параметр): из области видимости СИНИЙ (2), а не `studentName` из глобальной области видимости КРАСНЫЙ (1).

Когда вы решаете заместить переменную из внешней области видимости, одно из прямых следствий заключается в том, что из

этой области видимости вниз/вовнутрь (через все вложенные области видимости) переменная с этим именем уже не может быть отнесена к области видимости затененной переменной (КРАСНЫЙ (1) в данном случае). Иначе говоря, любая ссылка на идентификатор `studentName` будет соответствовать переменной-параметру и никогда — глобальной переменной `studentName`. Лексически невозможно сослаться на глобальную переменную `studentName` в какой-либо точке внутри функции `printStudent(..)` (или любой из вложенных областей видимости).

### Трюк с обратным затенением

Пожалуйста, учтите: пользоваться приемом, который я здесь опишу, не рекомендуется. Его полезность ограниченна, он запутывает читателей вашего кода и, скорее всего, приведет к появлению ошибок в программе. Я рассматриваю его только потому, что вы можете столкнуться с этим поведением в существующих приложениях, и лучший способ избежать путаницы — понять, что же происходит в программе.

К глобальной переменной можно обратиться из области видимости, в которой эта переменная была затенена, но обычно ссылки на лексический идентификатор будет недостаточно.

В глобальной области видимости (КРАСНЫЙ (1)) объявления `var` и объявления функций также предоставляются как свойства (с таким же именем, как у идентификатора) *глобального объекта* — по сути, объектного представления глобальной области видимости. Если вы писали код JS для браузера, вероятно, вы узнаете глобальный объект с именем `window`. Это не совсем точно, но достаточно для нашего обсуждения. В следующей главе тема глобальной области видимости/объекта будет рассмотрена более подробно.

Рассмотрим следующую программу, которая специально выполняется как автономный файл `.js` в среде браузера:

```
var studentName = "Suzy";

function printStudent(studentName) {
    console.log(studentName);
    console.log(window.studentName);
}

printStudent("Frank");
// "Frank"
// "Suzy"
```

Заметили ссылку `window.studentName`? Это выражение обращается к глобальной переменной `studentName` как свойству `window` (объект, который мы пока считаем синонимом глобального объекта). Это единственный способ обратиться к замещенной переменной из области видимости, в которой находится замещающая переменная.

Выражение `window.studentName` является синонимом глобальной переменной `studentName`, а не отдельной копией. Изменения в одной переменной будут отражаться в другой (в обоих направлениях). Выражение `window.studentName` можно рассматривать как альтернативный get/set-синтаксис для обращения к реальной переменной `studentName`. Собственно, вы даже можете *добавить* переменную в глобальную область видимости, создавая/задавая свойство глобального объекта.

Помните: то, что вы *можете* что-то сделать, вовсе не означает, что это *стоит* делать. Не затеняйте глобальную переменную, к которой нужно обращаться, и наоборот — по возможности не используйте этот прием для обращения к затененной глобальной переменной. И ни в коем случае не путайте читателей своего кода, создавая глобальные переменные как свойства window вместо формальных объявлений!

Этот маленький трюк подходит только для обращения к переменной из глобальной области видимости (не замещенной переменной из вложенной области видимости), и даже в этом случае только переменной, объявленной с ключевым словом `var` или `function`.

Другие формы объявлений в глобальной области видимости не создают зеркальных свойств глобального объекта:

```
var one = 1;
let notOne = 2;
const notTwo = 3;
class notThree {}

console.log(window.one); // 1
console.log(window.notOne); // undefined
console.log(window.notTwo); // undefined
console.log(window.notThree); // undefined
```

Переменные (как бы они ни были объявлены), существующие в любой другой области видимости, кроме глобальной, полностью недоступны в той области видимости, в которой они были замещены:

```
var special = 42;

function lookingFor(special) {
    // Идентификатор `special` (параметр) в этой
    // области видимости замещается внутри keepLooking()
    // и поэтому недоступен из этой области видимости.
    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(window.special);
    }
    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 42
```

Глобальная переменная `special` (КРАСНЫЙ (1)) замещается переменной из СИНИЙ (2) (параметр), а переменная `special` из СИНИЙ (2) сама замещается переменной `special` из ЗЕЛЕНЫЙ (3) внутри `keepLooking()`. К переменной `special` из КРАСНЫЙ (1) все еще можно обратиться по косвенной ссылке `window.special`. Однако `keepLooking()` никак не сможет обратиться к переменной `special` из области видимости СИНИЙ (2), содержащей число 112358132134.

## Копирование — не обращение

Следующий вопрос «но как насчет...?» мне задавали десятки раз. Пример:

```
var special = 42;

function lookingFor(special) {
    var another = {
        special: special
    };

    function keepLooking() {
        var special = 3.141592;
        console.log(special);
        console.log(another.special); // Все сложно!
        console.log(window.special);
    }

    keepLooking();
}

lookingFor(112358132134);
// 3.141592
// 112358132134
// 42
```

Выходит, другой объектный прием опровергает мое утверждение о том, что специальный параметр «полностью недоступен» из `keepLooking()`? Нет, утверждение остается истинным.

`special: special` копирует значение переменной-параметра `special` в другой контейнер (одноименное свойство). Конечно, если вы помещаете значение в другой контейнер, затенение уже не действует (если только оно тоже не было затенено). Но это не означает, что мы обращаемся к параметру `special`; это означает, что мы обращаемся к копии его значения на тот момент через *другой* контейнер (свойство объекта). Параметру `special` из СИНИЙ (2) не удастся присвоить другое значение из `keepLooking()`.

Другой вопрос из серии «но...?!», который у вас может возникнуть: а что, если использовать в качестве значений объекты или массивы

вместо чисел (112358132134 и т. д.)? Не решат ли ссылки на объекты вместо копий примитивных значений проблему недоступности?

Нет. Изменение содержимого объектного значения через копию ссылки — не то же самое, что лексическое обращение к самой переменной. Мы все равно не сможем присвоить другое значение параметру `special` из СИНИЙ (2).

### Недопустимое затенение

Не все комбинации замещения объявлений допустимы. `let` может замещать `var`, но `var` не может замещать `let`:

```
function something() {
    var special = "JavaScript";

    {
        let special = 42; // допустимое Затенение
        // ..
    }
}

function another() {
    // ..

    {
        let special = "JavaScript";

        {
            var special = "JavaScript";
            // ^^^ Синтаксическая ошибка
            // ..
        }
    }
}
```

Обратите внимание: в функции `another()` внутреннее объявление `var special` пытается объявить `special`, что само по себе нормально (как показывает функция `something()`).

Описание синтаксической ошибки в данном случае сообщает, что переменная `special` уже была определена, но это сообщение со-

держит дезинформацию. Еще раз: в `something()` такой ошибки нет, так как в замещении обычно ничего плохого нет.

Настоящая причина, по которой выдается ошибка `SyntaxError`, заключается в том, что `var` фактически пытается пересечь границу («перепрыгнуть») одноименного объявления `let`, что недопустимо. Запрет на пересечение границы фактически останавливается на каждой границе функции, так что в этом варианте исключение не возникает:

```
function another() {
    // ..

    {
        let special = "JavaScript";

        ajax("https://some.url",function callback(){
            // абсолютно нормальное Затенение
            var special = "JavaScript";

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

Подведем итог: объявление `let` (во внутренней области видимости) всегда может затенить объявление `var` во внешней области видимости. Объявление `var` (во внутренней области видимости) может затенить объявление `let` во внешней области видимости только в том случае, если между ними есть граница функции.

## Область видимости имени функции

Как вы уже видели, объявление функции выглядит примерно так:

```
function askQuestion() {
    // ..
}
```

Как обсуждалось в главах 1 и 2, такое объявление функции создает во внешней области видимости (в данном случае это гло-

бальная область видимости) идентификатор с именем `askQuestion`.

А как насчет следующей программы?

```
var askQuestion = function(){
    // ..
};
```

То же самое можно сказать о создаваемой переменной `askQuestion`. Но поскольку она является функциональным выражением (определением функции, используемым как значение вместо автономного объявления), сама функция не «поднимается» (см. главу 5).

У объявлений функций и функциональных выражений существует одно принципиальное отличие, связанное с идентификатором функции. Рассмотрим следующее функциональное выражение:

```
var askQuestion = function ofTheTeacher(){
    // ..
};
```

Мы знаем, что `askQuestion` в итоге оказывается во внешней области видимости. Но как насчет идентификатора `ofTheTeacher`? Для формальных объявлений функции идентификатор оказывается во внешней/вмещающей области видимости, поэтому может быть разумно предположить, что и здесь происходит то же. Но `ofTheTeacher` объявляется как идентификатор **внутри самой функции**:

```
var askQuestion = function ofTheTeacher() {
    console.log(ofTheTeacher);
};

askQuestion();
// function ofTheTeacher()...

console.log(ofTheTeacher);
// ReferenceError: переменная ofTheTeacher не определена
```

На самом деле не совсем точно утверждать, что `ofTheTeacher` находится *в области видимости функции*. В приложении А, раздел «Предполагаемые области видимости», эта тема будет рассмотрена более подробно.

Переменная `ofTheTeacher` не только объявляется внутри функции, а не снаружи, но и определяется как доступная только для чтения:

```
var askQuestion = function ofTheTeacher() {
    "use strict";
    ofTheTeacher = 42; // TypeError

    //..
};

askQuestion();
// TypeError
```

Так как мы использовали строгий режим, о неудачной попытке присваивания сообщается как об ошибке `TypeError`; в нестрогом режиме при подобных присваиваниях просто происходит сбой без выдачи исключения.

А что, если функциональное выражение не имеет идентификатора?

```
var askQuestion = function(){
   // ..
};
```

Функциональное выражение с идентификатором (именем) называется именованным функциональным выражением, а выражение без идентификатора называется анонимным функциональным выражением. Очевидно, у анонимных функциональных выражений нет идентификатора, влияющего на какую-либо из областей видимости.

Именованные и анонимные функциональные выражения рассматриваются намного подробнее (включая факторы, влияющие на решение об использовании одной или другой разновидности) в приложении А.

## Стрелочные функции

В ES6 в языке появилась дополнительная форма функциональных выражений — так называемые стрелочные функции:

```
var askQuestion = () => {
    // ..
};
```

Для определения функций `=>` ключевое слово `function` не требуется. Кроме того, в некоторых простых случаях круглые скобки `(..)` вокруг списка параметров не обязательны. Также в некоторых случаях фигурные скобки `{..}`, в которые заключается тело функции, не обязательны. И если `{..}` опускаются, возвращаемое значение передается без использования ключевого слова `return`.

К достоинствам стрелочных функций часто относится компактность синтаксиса; утверждается, что она равносильна созданию объективно более удобочитаемого кода. Это утверждение в лучшем случае сомнительно, а я считаю, что оно откровенно ошибочно. Удобочитаемость различных форм функций рассматривается в приложении А.

Стрелочные функции являются лексически анонимными; это означает, что в программе не существует напрямую связанного с ними идентификатора, который ссылается на функцию. Присваивание `askQuestion` создает автоматически определяемое имя `askQuestion`, но это не означает, что функция становится неанонимной:

```
var askQuestion = () => {
    // ..
};

askQuestion.name; // askQuestion
```

Стрелочные функции достигают своей синтаксической краткости за счет того, что вам приходится в уме жонглировать несколькими разновидностями разных форм/условий. Примеры:

```
() => 42;

id => id.toUpperCase();

(id,name) => ({ id, name });
```

```
(...args) => {
    return args[args.length - 1];
};
```

Настоящая причина, по которой я поднял тему стрелочных функций, — частые, но ошибочные утверждения о том, что стрелочные функции иначе ведут себя в отношении лексической области видимости по сравнению со стандартными функциями `function`.

Это не так.

Помимо анонимности (и отсутствия декларативной формы), стрелочные функции `=>` подчиняются тем же правилам лексической области видимости, что и функции `function`. Стрелочная функция (с фигурными скобками `{..}` вокруг ее тела или без них) все равно создает отдельную, внутреннюю вложенную область видимости. Объявления переменной в этой вложенной области видимости подчиняются тем же правилам, что и в функциональной области видимости.

## Итоги

При определении функции (объявления или выражения) создается новая область видимости. Расположение областей видимости, вложенных друг в друга, создает в программе естественную иерархию областей видимости, которая называется *цепочкой областей видимости*. Цепочка областей видимости управляет доступом к переменным.

Каждая новая область видимости предоставляет пространство для хранения ее собственного набора переменных. Если имя переменной повторяется на разных уровнях цепочки, происходит затенение, которое блокирует доступ к внешней переменной с этой точки по направлению к внутренним областям.

Впрочем, довольно технических подробностей. В следующей главе мы перейдем к главной области видимости, которая присутствует во всех программах JS, а именно к *глобальной области видимости*.

# 4 Глобальная область видимости

В главе 3 глобальная область видимости упоминалась несколько раз, но, возможно, вы все еще задаетесь вопросом, почему наружная область видимости программы играет такую важную роль в современном JS. Подавляющее большинство работы сейчас выполняется внутри функций и модулей, а не глобально.

Можно ли просто заявить: «Избегайте глобальной области видимости» — и считать, что этого достаточно?

Глобальная область видимости программы JS — богатая тема, которая содержит гораздо больше практической ценности и всевозможных нюансов, чем можно было бы ожидать. В этой главе мы сначала выясним, почему глобальная область видимости (все еще) полезна и актуальна для написания современных программ JS, а затем рассмотрим различия в том, где и *как обращаться* к глобальной области видимости в разных средах JS.

Полное понимание глобальной области видимости критично для вашего умения использовать лексические области видимости для формирования структуры ваших программ.

## Для чего нужна глобальная область видимости?

Вряд ли кого-нибудь из читателей удивит, что большинство приложений строится из нескольких (иногда многих) отдельных

файлов JS. Как же все эти раздельные файлы собираются воедино движком JS в одном контексте исполнения?

Относительно приложений, выполняемых в браузере, есть три основных способа.

Во-первых, если вы напрямую используете модули ES (без их транспиляции в другой формат упаковки модулей), эти файлы загружаются по отдельности средой JS. Затем каждый модуль импортирует ссылки на любые другие модули, к которым он должен обращаться. Файлы модулей взаимодействуют друг с другом исключительно через эти директивы импортирования, никакая общая внешняя область видимости им для этого не нужна.

Во-вторых, если вы используете упаковщик в своем процессе сборки приложения, все файлы обычно объединяются посредством конкатенации перед их передачей браузеру и движку JS, после чего обрабатывается только один большой файл. Даже когда все фрагменты приложения хранятся в одном файле, потребуется некий механизм, при помощи которого каждый фрагмент может зарегистрировать имя, к которому будут обращаться другие фрагменты, а также некоторые средства для реализации таких обращений.

В некоторых конфигурациях сборки приложений все содержимое файла упаковывается в одну внешнюю область видимости — например, функцию-обертку, универсальный модуль (UMD — см. приложение А) и т. д. Каждый фрагмент может зарегистрироваться для обращений со стороны других фрагментов, для чего используются локальные переменные этой общей области видимости. Пример:

```
(function wrappingOuterScope(){
    var moduleOne = (function one(){
        // ..
    })();

    var moduleTwo = (function two(){
        // ..

        function callModuleOne() {
            moduleOne.someMethod();
        }
```

```
        // ..
    })();
})();
```

Как видно из этого фрагмента, внутри области видимости функции `wrappingOuterScope()` объявляются локальные переменные `moduleOne` и `moduleTwo`, чтобы эти модули могли обращаться друг к другу для взаимодействия.

Хотя область видимости `wrappingOuterScope()` является функцией, а не полноценной средой глобальной области видимости, она работает как своего рода область видимости уровня приложения — область, в которой могут храниться все идентификаторы верхнего уровня (хотя и не является настоящей глобальной областью видимости.) В этом отношении она является своего рода заменителем глобальной области видимости.

И наконец, третий способ: независимо от того, используется ли для приложения программа-упаковщик или же файлы (кроме модулей ES) просто загружаются в браузере по отдельности (при помощи тегов `<script>` или других средств динамической загрузки ресурсов JS), если не существует одной охватывающей области видимости для всех этих фрагментов, глобальная область видимости становится единственным способом организации взаимодействия между ними.

Упакованный файл такого рода часто выглядит примерно так:

```
var moduleOne = (function one(){
    // ..
})();

var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

В данном случае окружающая область видимости функции отсутствует, поэтому объявления `moduleOne` и `moduleTwo` просто размещаются в глобальной области видимости. По сути это то же самое, как если бы файлы не объединялись конкатенацией, но загружались по отдельности:

```
module1.js:
var moduleOne = (function one(){
    // ..
})();

module2.js:
var moduleTwo = (function two(){
    // ..

    function callModuleOne() {
        moduleOne.someMethod();
    }

    // ..
})();
```

Если эти файлы загружаются по отдельности как обычные автономные файлы .js в среде браузера, каждое объявление переменной верхнего уровня оказывается глобальной переменной, так как глобальная область видимости оказывается единственным общим ресурсом для этих двух разных файлов — с точки зрения движка JS, они являются независимыми программами.

Кроме (потенциального) ведения учета того, где код приложения размещается во время выполнения, и возможностей каждого фрагмента по обращению к другим фрагментам для взаимодействия с ними в глобальной области видимости также:

- JS предоставляет доступ к своим встроенным
  - примитивам: `undefined`, `null`, `Infinity`, `NaN`;
  - платформенным средствам: `Date()`, `Object()`, `String()` и т. д.;
  - глобальным функциям: `eval()`, `parseInt()` и т. д.;
  - пространствам имен: `Math`, `Atomics`, `JSON`;
  - сопутствующим средствам: `Intl`, `WebAssembly`;

- среда, под управлением которой работает движок JS, предоставляет доступ к своим встроенным
  - консоли (и ее методам);
  - средствам DOM (`window`, `document` и т. д.);
  - таймерам (`setTimeout(..)` и т. д.);
  - API веб-платформы: `navigator`, `history`, геолокация, WebRTC и т. д.

И это далеко не все глобальные средства, с которыми может взаимодействовать ваша программа.

Node также предоставляет несколько элементов «глобально», но с технической точки зрения они не находятся в глобальной области видимости: `require()`, `__dirname`, `module`, URL и т. д.

Разработчики обычно согласны с тем, что глобальная область видимости не должна становиться свалкой для всех переменных вашего приложения. Свалка только создает хаос для многочисленных ошибок. Тем не менее нельзя отрицать, что глобальная область видимости становится важной *связующей средой* практически для любых приложений JS.

## Где именно находится глобальная область видимости?

Казалось бы, глобальная область видимости должна находиться во внешней части файла (т. е. не внутри какой-либо функции или другого блока). Но на самом деле все не так просто.

Разные среды JS по-разному обрабатывают области видимости вашей программы, особенно глобальную. У разработчиков JS часто встречаются заблуждения по этому поводу, даже если они не до конца понимают это.

## Объект window в браузере

В отношении глобальной области видимости самая чистая среда, в которой может работать JS, — это автономный файл .js, загружаемый в окружение веб-страницы в браузере. Под чистотой я имею в виду не то, что к ней ничего не добавляется (добавляться может очень многое), а скорее минимальное вмешательство в работу кода или ожидаемое поведение глобальной области видимости.

Рассмотрим следующий файл .js:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!
```

Этот код может быть загружен на веб-страницу при помощи встроенного тега `<script>`, тега `<script src=..>` в разметке или даже динамически созданного элемента DOM `<script>`. Во всех трех случаях идентификаторы `studentName` и `hello` объявляются в глобальной области видимости.

Это означает, что при обращении к глобальному объекту (обычно это объект `window` в браузере) вы найдете в нем одноименные свойства:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ window.studentName }!`);
}

window.hello();
// Hello, Kyle!
```

Это поведение по умолчанию, которое следует ожидать после чтения спецификации JS: внешняя область видимости является

глобальной, а `studentName` законно создается как глобальная переменная.

Вот что я имею в виду под чистотой. Но, к сожалению, это относится не ко всем средам JS, с которыми вы столкнетесь, и часто это оказывается неожиданным для разработчиков JS.

### Затенение глобальных имен глобальными именами

Вспомните описание затенения (и глобального обратного затенения) из главы 3: одно объявление переменной может переопределить объявление одноименной переменной из внешней области видимости и заблокировать доступ к ней.

Неожиданное последствие различий между глобальной переменной и одноименным глобальным свойством заключается в том, что внутри самой глобальной области видимости свойство глобального объекта может быть затенено глобальной переменной:

```
window.something = 42;

let something = "Kyle";

console.log(something);
// Kyle

console.log(window.something);
// 42
```

Объявление `let` добавляет глобальную переменную `something`, но не свойство глобального объекта (см. главу 3). В результате лексический идентификатор `something` затеняет свойство глобального объекта `something`.

Создавать расхождения между глобальным объектом и глобальной областью видимости почти всегда нежелательно. Они почти наверняка собьют с толку читателей вашего кода.

Все эти ловушки с глобальными объявлениями можно просто обойти: всегда используйте `var` для глобальных имен. `let` и `const` следует зарезервировать для блоковых областей видимости (см. раздел «Области видимости с блоками» главы 6).

## Глобальные средства DOM

Я утверждал, что среда JS под управлением браузера обладает самым чистым поведением глобальной области видимости, которая вам встретится. При этом она не является абсолютно чистой.

В приложениях JS на базе браузеров встречается один удивительный аспект поведения: элемент DOM с идентификатором `id` автоматически создает глобальную переменную, которая ссылается на него.

Возьмем следующую разметку:

```
<ul id="my-todo-list">
    <li id="first">Write a book</li>
    ..
</ul>
```

А код JS этой страницы может включать следующий фрагмент:

```
first;
// <li id="first">..</li>

    window["my-todo-list"];
// <ul id="my-todo-list">..</ul>
```

Если значение `id` является допустимым лексическим именем (таким как `first`), то будет создана лексическая переменная. Если нет, то обращение к этому глобальному элементу возможно только через глобальный объект (`window[..]`).

Автоматическая регистрация всех элементов DOM с идентификатором `id` — поведение, унаследованное от старых браузеров, которое должно оставаться, потому что многие старые сайты зависят от него. Мой совет: никогда не используйте эти глобальные переменные, хотя они всегда будут незаметно создаваться.

## Что в имени тебе моем?

Другая странность глобальной области видимости, встречающаяся в коде JS для браузеров:

```
var name = 42;

console.log(name, typeof name);
// "42" string
```

`window.name` — заранее определенный глобальный элемент в контексте браузера; это свойство глобального объекта, поэтому оно выглядит как нормальная глобальная переменная (однако ничего общего с «нормальностью» у него нет).

Мы использовали для объявления `var`, чтобы избежать затенения заранее определенного глобального свойства `name`. Фактически это означает, что объявление `var` игнорируется, так как у объекта глобальной области видимости уже есть свойство с таким именем. Как упоминалось ранее, если бы мы использовали `let name`, это привело бы к затенению `window.name` отдельной глобальной переменной `name`.

Но по-настоящему удивительно то, что, хотя мы присвоили `name` (а следовательно, и `window.name`) значение `42`, при получении значения вы получите строку `"42"`. В данном случае странность возникает из-за того, что `name` в действительности является заранее определенным get/set-свойством объекта, которое настаивает на том, что его значение является строкой. С ума сойти!

За исключением редких граничных случаев (таких как идентификатор элемента DOM и `window.name`), при выполнении JS из автономного файла в странице браузера наблюдается наиболее чистое поведение глобальной области видимости из тех, которые можно встретить.

## Веб-работники

Веб-работники (Web Workers) — расширение веб-платформы, работающее на базе поведения JS в браузере, что позволяет файлу

JS выполняться в потоке (уровня операционной системы), отдельном от потока, в котором выполняется основная программа JS.

Так как программы веб-работников выполняются в отдельном потоке, их взаимодействие с главным потоком приложения ограничивается для предотвращения/ограничения ситуаций гонки и других осложнений. Например, код веб-работников не имеет доступа к DOM. Однако некоторые веб-API доступны для работников (например, `navigator`).

Так как веб-работник рассматривается как совершенно отдельная программа, она не имеет общей глобальной области видимости с главной программой JS. Однако код все еще выполняется движком JS браузера, поэтому от поведения глобальной области видимости можно ожидать аналогичной *чистоты*. Так как модель DOM недоступна, синонима `window` для глобальной области видимости не существует.

В коде веб-работника обращение к глобальному объекту обычно осуществляется через `self`:

```
var studentName = "Kyle";
let studentID = 42;

function hello() {
    console.log(`Hello, ${ self.studentName }!`);
}

self.hello();
// Hello, Kyle!

self.studentID;
// undefined
```

Как и в главных программах JS, объявления `var` и `function` создают зеркальные свойства в глобальном объекте (он же `self`), тогда как другие объявления (`let` и т. д.) этого не делают.

Итак, наблюдаемое поведение глобальной области видимости практически не уступает по чистоте поведению, наблюдаемому при запуске программ JS; возможно, оно даже чище, потому что в нем отсутствует фактор DOM, затеняющий ситуацию!

## Консоль средств разработчика/REPL

Вспомните, о чем говорилось в главе 1 книги «Познакомьтесь, JavaScript»: средства разработчика не создают полностью интегрированной среды JS. Они обрабатывают код JS, но также склоняются в пользу UX-взаимодействий, наиболее дружественных к разработчикам (DX, Developer eXperience).

В некоторых случаях особое отношение к DX при вводе коротких фрагментов JS (вместо обычных формальных действий, ожидаемых при выполнении полной программы JS) создает наблюдаемые различия в поведении кода между программами и инструментами. Например, некоторые ситуации ошибок, действующие в программах JS, могут ослабляться и не отображаться при вводе кода в средствах разработчика. В контексте нашего обсуждения областей видимости к числу таких наблюдаемых различий могут принадлежать:

- поведение глобальной области видимости;
- поднятие (см. главу 5);
- конструкции объявлений в блоковой области видимости (`let`/`const`, см. главу 6) при использовании внешней области видимости.

И хотя может показаться, что при использовании консоли/REPL команды, введенные во внешней области видимости, обрабатываются в реальной глобальной области видимости, это не совсем точно. Такие инструменты обычно до какой-то степени эмулируют глобальную область видимости; это именно эмуляция, а не строгое соответствие. Инструментальные среды ставят на первое место удобство разработчика, а это означает, что в отдельных случаях (как в нашем текущем обсуждении областей видимости) наблюдаемое поведение может отклоняться от спецификации JS.

Из этого следует, что средства разработчика, оптимизированные для удобства и практичности операций разработки, не могут служить подходящими средами для определения или проверки явных и нетривиальных нюансов поведения в контексте реальных программ JS.

## Модули ES (ESM)

В ES6 появилась первоклассная поддержка паттерна «Модуль» (см. главу 8). Одно из самых очевидных следствий использования ESM связано с изменением поведения наблюдаемой области видимости верхнего уровня в файле.

Вспомните фрагмент кода, приводившийся ранее (который будет приведен к формату ESM ключевым словом `export`):

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

export hello;
```

Если этот код хранится в файле, который загружается как модуль ES, он будет работать точно так же. Но наблюдаемые эффекты с точки зрения приложения в целом будут другими. Несмотря на объявление на верхнем уровне файла (модуля), в наружной области видимости `studentName` и `hello` не являются глобальными переменными. Они существуют на уровне модуля или, если хотите, являются модульно-глобальными.

Однако в модуле не существует неявного «объекта области видимости уровня модуля», к которому эти объявления верхнего уровня добавлялись бы в виде свойств, как это происходит при размещении объявлений на верхнем уровне файлов JS, не являющихся модулями. Это не означает, что в таких программах не могут существовать глобальные переменные или к ним невозможно обратиться. Просто глобальные переменные не создаются объявлением переменных в области верхнего уровня модуля.

Область видимости верхнего уровня в модуле происходит от глобальной области видимости почти так же, как если бы все со-

держимое модуля было упаковано в функцию. Таким образом, все переменные, существующие в глобальной области видимости (независимо от того, содержатся они в глобальном объекте или нет!), доступны в виде лексических идентификаторов из области видимости модуля.

ESM способствует минимизации зависимости от глобальной области видимости: вы импортируете те модули, которые необходимы для функционирования текущего модуля. Как следствие, в коде реже встречаются случаи использования глобальной области видимости или ее глобального объекта.

Тем не менее, как упоминалось ранее, остается еще много глобальных элементов JS и веб-элементов, к которым вы будете по-прежнему обращаться из глобальной области видимости независимо от того, сознаете вы это или нет.

## Node

У Node есть одна особенность, которая часто застает врасплох разработчиков JS: Node рассматривает каждый загруженный отдельный файл .js, включая главный файл, из которого запускается процесс Node, как модуль (модуль ES или модуль CommonJS, см. главу 8). Практическое следствие заключается в том, что верхний уровень программ Node никогда не бывает глобальной областью видимости при загрузке в браузере файлов, не являющихся модулями.

На момент написания книги в Node недавно появилась поддержка модулей ES. Но кроме этого, в Node с самого начала поддерживался формат модулей, который обычно называется CommonJS и выглядит так:

```
var studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}
```

```
hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Прежде чем продолжать, Node фактически упаковывает такой код в функцию, чтобы объявления `var` и `function` содержались в области видимости функции-обертки, а не считались глобальными переменными.

Можно считать, что Node воспринимает приведенный код в следующем виде (приводится для пояснения, не как реальный код):

```
function Module(module,require,__dirname,...) {
    var studentName = "Kyle";

    function hello() {
        console.log(`Hello, ${ studentName }!`);
    }

    hello();
    // Hello, Kyle!

    module.exports.hello = hello;
}
```

Затем Node фактически вызывает добавленную функцию `Module(..)` для запуска вашего модуля. Здесь четко видно, почему идентификаторы `studentName` и `hello` не являются глобальными, а объявлены в области видимости модуля.

Как упоминалось ранее, Node определяет ряд глобальных элементов (таких как `require()`), но в действительности они не являются идентификаторами в глобальной области видимости (или свойствами глобального объекта). Они внедряются в область видимости каждого модуля, отчасти напоминая параметры, перечисленные в объявлении функции `Module(..)`.

Как же определять реальные глобальные переменные в Node? Это можно сделать только одним способом: добавлением свойств в другой автоматически представляемый глобальный элемент Node, который, как ни парадоксально, называется `global`. `global`

содержит ссылку на реальный объект глобальной области видимости — нечто вроде `window` в среде JS браузера.

Пример:

```
global.studentName = "Kyle";

function hello() {
    console.log(`Hello, ${ studentName }!`);
}

hello();
// Hello, Kyle!

module.exports.hello = hello;
```

Здесь мы добавляем `studentName` как свойство объекта `global`, после чего в команде `console.log(..)` к `studentName` можно обращаться как к обычной глобальной переменной.

Помните, что JS не определяет идентификатор `global`; он определяется именно Node.

## globalThis

Кратко резюмируя среды JS, рассмотренные до настоящего момента, программа может делать (или не делать) следующее:

- объявить глобальную переменную в области видимости верхнего уровня с использованием объявлений `var` и `function` или `let`, `const` и `class`;
- также добавлять объявления глобальных переменных как свойства объекта глобальной области видимости, если для объявления используется ключевое слово `var` или `function`;
- обращаться к объекту глобальной области видимости (для добавления и получения глобальных переменных как свойств) через `window`, `self` или `global`.

Думаю, можно с полным основанием сказать, что поведение глобальной области видимости и обращения к ней сложнее, чем

считает большинство разработчиков и как было показано в предыдущих разделах. Однако сложность нигде не проявляется так очевидно, как при попытке получить универсально применимую ссылку на объект глобальной области видимости.

Еще один трюк для получения ссылки на объект глобальной области видимости выглядит так:

```
const theGlobalScopeObject =
    (new Function("return this"))();
```

Функция может быть динамически сконструирована из кода, хранящегося в строковом значений, конструктором `Function()`, сходным с `eval(..)` (см. раздел «Изменение области видимости во время выполнения», глава 1). Такая функция будет автоматически выполняться в нестрогом режиме (по соображениям совместимости) при нормальном вызове с использованием механизма `()`; `this` в ней будет указывать на глобальный объект. За дополнительной информацией об определении таких привязок обращайтесь к третьей книге серии, «Объекты и классы».

Итак, у нас есть `window`, `self`, `global` и новый безобразный трюк с `new Function(..)`. Много разных способов получения глобального объекта. У каждого есть свои достоинства и недостатки.

Так почему не добавить еще один?!

По состоянию на ES2020 в JS наконец-то появилась стандартизированная ссылка на объект глобальной области видимости, которая называется `globalThis`. Итак, в зависимости от новизны движок JS, выполняющих ваш код, вместо всех перечисленных подходов можно использовать `globalThis`.

Можно даже попытаться определить межсредовое полизаполнение, более надежно работающее в средах JS до появления `globalThis`:

```
const theGlobalScopeObject =
    (typeof globalThis != "undefined") ? globalThis :
    (typeof global != "undefined") ? global :
    (typeof window != "undefined") ? window :
    (typeof self != "undefined") ? self :
    (new Function("return this"))();
```

Ух! Конечно, такое решение неидеально, но оно работает, если вам потребуется надежная ссылка на глобальную область видимости.

Предложенное имя `globalThis` вызвало ожесточенные споры при добавлении этой возможности в JS. А именно я и многие другие считали, что ссылка `this` в имени создает неверное впечатление, потому что вы обращаетесь по ссылке на этот объект для того, чтобы получить доступ к глобальной области видимости, и никогда — для обращения к некоторой разновидности привязки `this` (глобальной/по умолчанию). Также рассматривалось много других имен, но по различным причинам они были отвергнуты. К сожалению, выбранное имя оказалось последним выходом. Если вы собираетесь взаимодействовать с объектами глобальной области видимости в своих программах, то для предотвращения путаницы я настоятельно рекомендую выбрать более понятное имя — например, использованное в данном примере имя `theGlobalScopeObject` (смехотворно длинное, но точное!).

## Глобальная осведомленность

Глобальная область видимости присутствует и остается актуальной в каждой программе JS, несмотря на то что современные паттерны организации кода по модулям в значительной мере сокращают зависимость от хранения идентификаторов в этом пространстве имен.

Но по мере того как наш код все дальше выходит за рамки браузера, особенно важно хорошо представлять различия в глобальной области видимости (и объекте глобальной области видимости!) между разными средами JS.

Общая картина глобальной области видимости немного прояснилась, и в следующей главе мы снова погрузимся в технические подробности лексической видимости: как и когда могут использоваться переменные.

# 5 (Не такой уж) тайный жизненный цикл переменных

Теперь вы должны достаточно хорошо понимать концепцию вложенности от глобальной области видимости вниз — так называемой *цепочки областей видимости* программы.

Но просто знать, из какой области видимости происходит переменная, недостаточно. Если объявление переменной следует после первой команды в области видимости, как будут вести себя ссылки на этот идентификатор до его объявления? Что произойдет, если попытаться дважды объявить одну переменную в области видимости?

Особая разновидность лексических областей видимости в JS богата нюансами в том, как и когда переменные начинают свое существование и становятся доступными для программы.

## Когда можно использовать переменную?

В какой момент переменная становится доступной для использования внутри своей области видимости? Казалось бы, ответ очевиден: *после* того, как переменная была объявлена/создана. Правильно? Не совсем.

Пример:

```
greeting();
// Hello!

function greeting() {
    console.log("Hello!");
}
```

Этот код работает нормально. Возможно, вы уже видели или даже писали нечто подобное. Но вы когда-нибудь задумывались над тем, как и почему он работает? А если конкретно, почему вы можете обратиться к идентификатору `greeting` из строки 1 (для получения и выполнения ссылки на функцию), хотя объявление функции `greeting()` происходит только в строке 4?

Вспомните: в главе 1 я говорил, что все идентификаторы регистрируются в соответствующих областях видимости во время компиляции. Более того, каждый идентификатор создается в начале той области видимости, которой он принадлежит, при каждом входе в эту область видимости.

Ситуация, при которой переменная видима от начала своей вмещающей области видимости, несмотря на то что ее объявление может находиться ниже в области видимости, называется *поднятием* (hoisting).

Тем не менее одно лишь поднятие не дает полного ответа на вопрос. Идентификатор `greeting` виден от начала области видимости, но почему мы можем использовать функцию `greeting()` еще до того, как она была объявлена?

Другими словами, как переменной `greeting` может быть присвоено значение (ссылка на функцию) с момента выполнения области видимости? Ответ кроется в специальной характеристике формальных объявлений `function`, называемой *поднятием функций*. Когда идентификатор объявления `function` регистрируется в начале своей области видимости, он дополнительно автоматически

инициализируется ссылкой на эту функцию. Вот почему функция может вызываться во всей области видимости!

Одна ключевая подробность заключается в том, что и *поднятие функции*, и *поднятие* `var`*-переменных* связывают свои идентификаторы с ближайшей вмещающей **областью видимости функции** (а если ее нет — глобальной областью видимости), а не с блоковой областью видимости.

Объявления с `let` и `const` тоже поднимаются (см. часть о TDZ далее в этой главе). Но эти две формы объявлений связываются со своим вмещающим блоком, а не с вмещающей функцией, как в случае с объявлениями `var` и `function`. За дополнительной информацией обращайтесь к разделу «Области видимости и блоки» главы 6.

## Поднятие: объявления и выражения

*Поднятие функций* применяется только к формальным объявлениям функций (а конкретно тех, которые располагаются вне блоков — см. раздел FiB главы 6), но не к присваиваниям функциональных выражений.

Пример:

```
greeting();
// TypeError

var greeting = function greeting() {
    console.log("Hello!");
};
```

В строке 1 (`greeting();`) происходит ошибка. И очень важно заметить *разновидность* этой ошибки. `TypeError` означает, что мы пытаемся что-то сделать с недопустимым значением. В зависимости от вашей среды JS в сообщении об ошибке может быть сказано что-то вроде «'undefined' не является функцией» или в более содержательном варианте — «'greeting' не является функцией».

Обратите внимание: программа не выдает ошибку `ReferenceError`. JS не говорит, что найти идентификатор в области видимости не удалось. JS говорит, что идентификатор `greeting` был найден, но в данный момент он не содержит ссылки на функцию. Вызываться могут только функции, поэтому попытка вызова для любого значения, не являющегося функцией, приводит к ошибке.

Но что же содержит `greeting`, если не ссылку на функцию? Кроме поднятия, переменные, объявленные с ключевым словом `var`, также автоматически инициализируются `undefined` в начале своей области видимости — это снова ближайшая вмещающая или глобальная область видимости. После инициализации они доступны для использования (присваивания, чтения и т. д.) во всей области видимости.

Таким образом, в первой строке `greeting` существует, но содержит только значение по умолчанию `undefined`. Только в строке 4 `greeting` будет присвоена ссылка на функцию.

Обратите особое внимание на это отличие. Объявление функции поднимается и **инициализируется своим значением функции** (еще раз: это называется *поднятием функции*). Переменная `var` тоже поднимается и автоматически инициализируется `undefined`. Все последующие присваивания функциональных выражений этой переменной не выполняются до того момента, когда это присваивание будет обработано во время выполнения.

В обоих случаях идентификатор (имя) поднимается. Но связывание ссылки на функцию не обрабатывается во время инициализации (начало области видимости), если только идентификатор не был создан в формальном объявлении `function`.

## Поднятие переменной

Рассмотрим еще один пример поднятия переменной:

```
greeting = "Hello!";
console.log(greeting);
// Hello!

var greeting = "Howdy!";
```

И хотя идентификатор `greeting` не объявляется до строки 4, он доступен для присваивания уже в строке 1. Почему?

У объяснения есть две необходимые составляющие:

- идентификатор поднимается;
- он автоматически инициализируется значением `undefined` в начале области видимости.

Вероятно, подобное использование *поднятия переменных* выглядит неестественно, и многие читатели с полным основанием предпочтут избегать его в своих программах. Но следует ли избегать всего поднятия (включая поднятие функций)? Разные точки зрения на поднятие будут более подробно рассмотрены в приложении А.

## Поднятие: еще одна метафора

Глава 2 была полна метафор (для демонстрации областей видимости), но здесь мы сталкиваемся еще с одной — самим поднятием (hoisting). Не будем представлять поднятие как конкретный шаг, выполняемый движком JS на стадии выполнения, а рассмотрим его как наглядное представление различных действий, выполняемых движком JS при подготовке программы **до выполнения**.

Как правило, термин «поднятие» вызывает ассоциации с поднятием тяжестей — т. е. любых идентификаторов до самого верха области видимости. В объяснениях часто утверждается, что движок JS фактически перезаписывает программу перед выполнением, так что в результате она выглядит примерно так:

```
var greeting;          // поднятое объявление
greeting = "Hello!";   // исходная строка 1
console.log(greeting); // Hello!
greeting = "Howdy!";   // `var` исчезает!
```

Поднятие (метафора) предполагает, что JS осуществляет предварительную обработку исходной программы и немного переупорядочивает его, так что все объявления перемещаются в начало соответствующих областей видимости перед выполнением. Более того, метафора поднятия утверждает, что все объявления функций полностью поднимаются в начало своих областей видимости. Пример:

```
studentName = "Suzy";
greeting();
// Hello Suzy!

function greeting() {
    console.log(`Hello ${ studentName }!`);
}
var studentName;
```

«Правило» метафоры поднятия гласит, что объявления функций поднимаются первыми, а затем после всех функций немедленно поднимаются переменные. Таким образом, поднятие предполагает, что программа *переупорядочивается* движком JS так, чтобы она пришла к следующему виду:

```
function greeting() {
    console.log(`Hello ${ studentName }!`);
}
var studentName;

studentName = "Suzy";
greeting();
// Hello Suzy!
```

Метафора поднятия удобна. Она позволяет выполнить опережающую предварительную обработку, необходимую для нахождения всех этих объявлений, зарытых где-то глубоко в областях видимости, и каким-то волшебным образом поднять (переместить) их в начало; всю программу можно рассматривать так, словно она выполняется движком JS сверху вниз за один проход.

Однопроходная обработка определенно выглядит более понятно, чем двухфазная обработка, упоминавшаяся в главе 1.

Поднятие как механизм переупорядочения кода — привлекательное упрощенное представление, но оно неточно. Движок JS не осуществляет фактической перестановки кода. Оно не может по волшебству заглянуть вперед и найти объявления; точно определить их (а также все границы областей видимости в программе) можно только одним способом — полным разбором кода.

Угадайте, о каком разборе идет речь? О первой фазе двухфазной обработки! И никакие словесные ухищрения не обойдут этот факт.

Таким образом, если метафора поднятия (в лучшем случае) неточна, что же делать с этим термином? Я думаю, что он остается полезным — даже участники TC39 регулярно пользуются им! — но мы не можем утверждать, что он представляет фактическое переупорядочение исходного кода.

Неправильные или неполные ментальные модели часто кажутся адекватными, потому что они в отдельных случаях приводят к правильным ответам. Но в долгосрочной перспективе вам будет труднее анализировать и прогнозировать результаты, если ваши внутренние представления расходятся с тем, как реально работает движок JS.

Я считаю, что термин «поднятие» *следует* использовать для обозначения выполняемой во время компиляции операции генерирования команд для автоматической регистрации переменной в начале ее области видимости при каждом входе в эту область видимости.

Это тонкий, но важный переход от поднятия как поведения времени выполнения до его более подобающего места среди задач времени компиляции.

## Повторное объявление?

Как вы думаете, что произойдет при многократном объявлении переменной в той же области видимости? Пример:

```
var studentName = "Frank";
console.log(studentName);
// Frank

var studentName;
console.log(studentName); // ???
```

Что, по вашему мнению, будет выведено во втором сообщении? Многие разработчики считают, что второе вхождение `var studentName` объявляет переменную заново (и при этом «сбрасывает» ее), поэтому команда должна вывести `undefined`.

Но существует ли такое явление, как повторное объявление переменной в одной области видимости? Нет.

Если рассматривать программу с точки зрения метафоры поднятия, этот код можно было бы переупорядочить в следующем виде для целей выполнения:

```
var studentName;
var studentName; // очевидно бессмысленная пустая операция!

studentName = "Frank";
console.log(studentName);
// Frank

console.log(studentName);
// Frank
```

Так как суть поднятия в действительности заключается в регистрации переменной в начале области видимости, если исходная программа содержит вторую команду `var studentName`, в середине области видимости ничего делать не придется. По сути это пустая операция; такая команда не имеет смысла.

Если продолжить метафору диалога из главы 2, *Компилятор* найдет вторую команду `var` и спросит *Менеджера области видимости*, встречался ли ему идентификатор `studentName`; так как идентификатор уже встречался, ничего делать не нужно.

Также важно заметить, что `var studentName;` не означает `var studentName = undefined;`, как считают многие. Чтобы доказать, что это не одно и то же, рассмотрим следующую разновидность программы:

```
var studentName = "Frank";
console.log(studentName); // Frank

var studentName;
console.log(studentName); // Frank <--- все равно!

// добавим явную инициализацию
var studentName = undefined;
console.log(studentName); // undefined <--- видите!?
```

Как видите, явная инициализация `= undefined` дает другой результат, чем предположение о том, что она выполняется неявно при ее отсутствии. В следующем разделе мы вернемся к теме инициализации переменных по их объявлениям.

Повторное объявление `var` с тем же именем идентификатора в области видимости фактически является пустой операцией. Другая иллюстрация — на этот раз через функцию с тем же именем:

```
var greeting;

function greeting() {
    console.log("Hello!");
}

// по сути пустая операция
var greeting;

typeof greeting; // "function"

var greeting = "Hello!";

typeof greeting; // "string"
```

Первое объявление `greeting` регистрирует идентификатор в области видимости, а так как это объявление `var`, оно будет автоматически инициализировано `undefined`. Объявлению `function` не нужно заново регистрировать идентификатор, но из-за *поднятия*

*функций* оно переопределяет автоматическую инициализацию для использования ссылки на функцию. Второе объявление `var greeting` само по себе ничего не делает, потому что `greeting` уже является идентификатором, а поднятие функции уже имеет приоритет перед автоматической инициализацией.

На самом деле присваивание `"Hello!"` переменной `greeting` заменяет ее значение (исходную функцию `greeting()`) строкой; само по себе объявление `var` никакого эффекта не имеет.

Как насчет повторения объявлений в области видимости с использованием `let` или `const`?

```
let studentName = "Frank";

console.log(studentName);

let studentName = "Suzy";
```

Программа не будет выполняться, но вместо этого немедленно выдаст ошибку `SyntaxError`. В зависимости от среды JS сообщение об ошибке может включать сообщение вида «имя studentName уже было объявлено». Иначе говоря, это тот случай, когда попытка повторного объявления явно запрещена!

Дело даже не в том, что с двумя объявлениями, использующими `let`, произойдет эта ошибка. Если хотя бы одно объявление использует `let`, в другом может использоваться `let` или `var`, и ошибка все равно произойдет, как показывают следующие два примера:

```
var studentName = "Frank";

let studentName = "Suzy";
```

и:

```
let studentName = "Frank";

var studentName = "Suzy";
```

В обоих случаях `SyntaxError` выдается для *второго* объявления. Иначе говоря, повторно объявить переменную можно только

одним способом — использовать `var` во всех (двух или более) объявлениях.

Но почему это запрещено? Причина выдачи ошибки не является чисто технической, так как повторное объявление с `var` всегда было разрешено; очевидно, то же самое можно было сделать и для `let`. Скорее дело в социотехнике. Повторное объявление переменных рассматривается многими, включая участников комитета TC39, как вредная привычка, которая может привести к появлению ошибок в программах. Таким образом, когда в ES6 появилась поддержка `let`, было решено предотвратить повторное объявление при помощи ошибки.

Конечно, это всего лишь стилистическое предпочтение, а не технический аргумент. Многие разработчики согласны с этой позицией, и, скорее всего, это отчасти объясняет, почему в TC39 было решено включить эту ошибку (а также поведение `let` в соответствии с `const`). Однако также можно было выдвинуть разумный аргумент о том, что сохранение согласованности с прецедентом `var` было более благоразумно и что такие принудительные меры лучше было оставить для инструментов, применяемых по усмотрению разработчика (например, синтаксических анализаторов). В приложении A мы рассмотрим вопрос о том, остается ли объявление `var` (и связанное с ним поведение — например, повторное объявление) полезным в современном JS.

Когда *Компилятор* обращается к *Менеджеру области видимости* с запросом об объявлении (был ли идентификатор объявлен ранее) и если в одном/обоих объявлениях используется `let`, происходит ошибка. Она должна подать разработчику четкий сигнал: «Не надо полагаться на некорректные переобъявления!»

### Константы?

Ключевое слово `const` создает больше ограничений, чем `let`. Как и `let`, `const` не может повторяться с тем же идентификатором в одной области видимости. Однако в данном случае существует

важная техническая причина, по которой запрещаются подобные повторные объявления (в отличие от ситуации с `let`, где повторное объявление запрещается в основном по стилистическим причинам).

Ключевое слово `const` требует, чтобы переменная была инициализирована, так что если опустить присваивание в объявлении, это приведет к ошибке `SyntaxError`:

```
const empty; // SyntaxError
```

Объявления `const` создают переменные, которым не может быть присвоено другое значение:

```
const studentName = "Frank";
console.log(studentName);
// Frank

studentName = "Suzy"; // TypeError
```

Переменной `studentName` нельзя присвоить новое значение, потому что она объявлена с ключевым словом `const`.

При попытке повторного присваивания `studentName` выдается ошибка `TypeError`, а не `SyntaxError`. Это довольно важный нюанс, который, к сожалению, очень легко упустить из вида. Ошибки `SyntaxError` представляют сбои в программе, которые не позволяют начать выполнение. Ошибки `TypeError` представляют аномалии, возникающие в ходе выполнения программы. В приведенном фрагменте сообщение `"Frank"` выводится до обработки повторного присваивания `studentName`, при котором происходит ошибка.

Таким образом, если повторное присваивание для объявлений `const` невозможно, а объявления `const` всегда требуют присваивания, появляется четкая техническая причина, по которой для `const` должны быть запрещены повторные объявления: любое повторное объявление `const` также неизбежно станет повторным присваиванием `const`, а это запрещено!

```
const studentName = "Frank";

// очевидно, должно приводить к ошибке
const studentName = "Suzy";
```

Так как повторное объявление для `const` должно быть запрещено (по техническим причинам), комитет TC39 в итоге решил, что «повторное объявление» для `let` также должно быть запрещено для обеспечения целостности. Трудно сказать, было ли это лучшим решением, но, по крайней мере, оно было продиктовано внутренней логикой.

### Циклы

Из предыдущего обсуждения должно быть ясно, что JS на самом деле не хочет, чтобы мы «повторно объявляли» переменные в пределах области видимости.

Возможно, это кажется банальной мерой предосторожности, пока вы не задумаетесь, что же означает повторное выполнение команд объявления в циклах. Пример:

```
var keepGoing = true;
while (keepGoing) {
    let value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Переменная `value` многократно повторно объявляется в программе? Приводит ли это к выдаче ошибок? Нет.

Все правила области видимости (включая повторное объявление переменных, созданных с `let`) применяются *на уровне экземпляра области видимости*. Иначе говоря, каждый раз, когда во время выполнения программа входит в область видимости, происходит сброс в исходное состояние.

Каждая итерация цикла имеет собственный новый экземпляр области видимости, и в каждом экземпляре области видимости

value объявляется только один раз. Таким образом, попыток повторного объявления нет, а следовательно, нет и ошибки. Прежде чем рассматривать другие формы циклов, что произойдет, если заменить объявление value в приведенном фрагменте на var?

```
var keepGoing = true;
while (keepGoing) {
    var value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Происходит ли повторное объявление переменной value — ведь мы знаем, что var его допускает? Нет. Так как переменная var не рассматривается как объявление с блоковой областью видимости (см. главу 6), она присоединяется к глобальной области видимости. Таким образом, здесь существует только одна переменная value в одной области видимости с keepGoing (глобальная область видимости в данном случае). Никакого повторного объявления здесь тоже нет!

Чтобы разобраться во всем этом, можно запомнить, что ключевые слова var, let и const фактически *удаляются* из кода к тому моменту, когда он начинает выполняться. Всем этим занимается исключительно компилятор.

Если вы мысленно удалите ключевые слова объявлений, а потом попытаетесь прочитать код, это поможет вам решить, где могут возникнуть (повторные) объявления (и могут ли вообще).

Как насчет повторного объявления в других разновидностях циклов — например, циклов for?

```
for (let i = 0; i < 3; i++) {
    let value = i * 10;
    console.log(`${ i }: ${ value }`);
}
// 0: 0
// 1: 10
// 2: 20
```

Пример наглядно показывает, что на каждый экземпляр области видимости объявляется отдельный экземпляр `value`. Но как насчет `i`? Эта переменная объявляется повторно?

Чтобы ответить на этот вопрос, посмотрим, в какой области видимости находится переменная `i`. Может показаться, что она принадлежит внешней (в данном случае глобальной) области видимости, но это не так. Она находится в области видимости тела цикла `for`, как и `value`. Этот цикл можно неформально рассматривать в следующей эквивалентной форме:

```
{
    // фиктивная переменная для демонстрации
    let $$i = 0;

    for ( /* ничего */; $$i < 3; $$i++) {
        // настоящий цикл `i`!
        let i = $$i;

        let value = i * 10;
        console.log(`${ i }: ${ value }`);
    }
    // 0: 0
    // 1: 10
    // 2: 20
}
```

Сейчас должно быть ясно: переменные `i` и `value` объявляются ровно **один раз на экземпляр области видимости**. Повторных объявлений здесь нет.

Как насчет других форм циклов `for`?

```
for (let index in students) {
    // нормально
}
for (let student of students) {
    // как и это
}
```

Сказанное относится к циклам `for..in` и `for..of`: объявленная переменная рассматривается как находящаяся внутри тела цикла,

а следовательно, обрабатывается на уровне итерации (т. е. на экземпляр области видимости). Снова никаких повторных объявлений.

Наверное, сейчас я уже напоминаю вам заезженную пластинку. Но давайте посмотрим, как `const` влияет на циклические конструкции. Пример:

```
var keepGoing = true;
while (keepGoing) {
    // новенькая константа!
    const value = Math.random();
    if (value > 0.5) {
        keepGoing = false;
    }
}
```

Как и в варианте программы с `let`, приведенном выше, объявление `const` выполняется ровно один раз для каждой итерации цикла, поэтому оно защищено от проблем с повторным объявлением. Но когда речь заходит о циклах `for`, ситуация усложняется.

Циклы `for..in` и `for..of` вполне нормально работают с `const`:

```
for (const index in students) {
    // нормально
}

for (const student of students) {
    // как и это
}
```

Но не общий цикл `for`:

```
for (const i = 0; i < 3; i++) {
    // после первой итерации происходит
    // сбой с ошибкой TypeError
}
```

Что здесь не так? В этой конструкции можно было нормально использовать `let`, и ранее было сказано, что для области видимости каждой итерации создается новое значение `i`, потому вроде бы никакого повторного объявления здесь нет.

Мысленно расширим этот цикл, как это делалось ранее:

```
{
    // фиктивная переменная для демонстрации
    const $$i = 0;

    for ( ; $$i < 3; $$i++) {
        // настоящий цикл `i`!
        const i = $$i;
        // ..
    }
}
```

Заметили проблему? Переменная `i` действительно просто создается за пределами цикла. С ней проблем нет. Проблема кроется в концептуальной переменной `$$i`, которая должна каждый раз инкрементироваться выражением `$$i++`. Это повторное присваивание (не повторное объявление), которое запрещено для констант.

Помните, что эта расширенная форма — не более чем концептуальная модель, которая помогает вам понять источник проблемы. Возможно, вас интересует, не может ли JS преобразовать `const $$i = 0` в `let $ii = 0`, что позволило бы `const` работать с классическим циклом `for`? Да, такое возможно, но это создало бы потенциально неожиданные исключения в семантике цикла `for`.

Например, возникло бы произвольное (и скорее всего, непонятное) неочевидное исключение: выражению `i++` в заголовке цикла `for` было бы разрешено обходить жесткие ограничения с присваиванием `const`, тогда как другие повторные присваивания `i` внутри итерации цикла были бы запрещены (хотя иногда они бывают полезными).

Наиболее прямолинейный ответ: `const` не может использоваться с классической формой `for`-цикла из-за необходимого повторного присваивания. Интересно, что без повторного присваивания такая конструкция действительна:

```
var keepGoing = true;

for (const i = 0; keepGoing; /* здесь ничего нет */ ) {
    keepGoing = (Math.random() > 0.5);
    // ..
}
```

Этот фрагмент работает, но смысла в нем нет. Бессмысленно объявлять `i` в этой позиции с ключевым словом `const`, так как переменная в этой позиции объявляется именно для подсчета итераций. Просто используйте другую форму цикла (например, цикл `while`) или `let`!

## Неинициализированные переменные (TDZ)

С объявлениями `var` переменная поднимается в начало своей области видимости. Но она также автоматически инициализируется значением `undefined`, так что переменная может использоваться во всей области видимости.

Но объявления `let` и `const` в этом отношении ведут себя не совсем одинаково.

Пример:

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

let studentName = "Suzy";
```

В результате выполнения этой программы в первой строке выдается ошибка `ReferenceError`. В зависимости от вашей среды JS в сообщении об ошибке может быть сказано что-то вроде «не удается обратиться к `studentName` до инициализации».

Приведенное сообщение об ошибке когда-то было намного более невразумительным или дезинформирующим. К счастью, некоторым участникам сообщества удалось добиться улучшения этого сообщения в движках JS, чтобы оно более точно сообщало вам, что произошло в программе!

Это сообщение достаточно четко сообщает, что не так: переменная `studentName` существует в строке 1, но она не была инициализиро-

вана и поэтому использоваться еще не может. Попробуем сделать это:

```
studentName = "Suzy"; // попробуем инициализировать!
// ReferenceError

console.log(studentName);

let studentName;
```

Увы. Все равно возникает ошибка `ReferenceError`, но на этот раз в первой строке, где мы пытаемся присвоить значение (т. е. инициализировать) так называемую неинициализированную переменную `studentName`. Как это понимать?!

Настоящий вопрос в том, как инициализировать неинициализированную переменную? Для `let`/`const` это можно сделать только присваиванием, совмещенным с командой объявления. Одного присваивания недостаточно! Пример:

```
let studentName = "Suzy";
console.log(studentName); // Suzy
```

Здесь переменная `studentName` инициализируется (в данном случае `"Suzy"` вместо `undefined`) посредством разновидности объявления `let`, совмещенной с присваиванием.

Другое возможное решение:

```
// ..

let studentName;
// or:
// let studentName = undefined;

// ..

studentName = "Suzy";

console.log(studentName);
// Suzy
```

Интересно! Вспомните: ранее мы говорили, что `var studentName;` — не то же самое, что `var studentName = undefined;`, но в данном случае с `let` они ведут себя одинаково. Различия сводятся к тому, что `var studentName` автоматически инициализируется в начале области видимости, а `let studentName` — нет.

Ранее мы уже несколько раз говорили, что компилятор в конечном итоге удаляет все объявления `var`/`let`/`const` и заменяет их командами регистрации соответствующих идентификаторов в начале каждой области видимости.

Итак, если проанализировать, что здесь происходит, мы видим, что компилятор также добавляет команду в середине программы, в точке объявления переменной `studentNamewas`, для обеспечения автоматической инициализации этого объявления. Эта переменная не может использоваться в любой точке, предшествующей этой инициализации. Все сказанное относится к `const` в такой же степени, как `let`.

Для обозначения этого периода от входа в область видимости до точки автоматической инициализации переменной в TC39 был предложен термин TDZ (Temporal Dead Zone).

TDZ — временное окно, в котором переменная существует, но все еще остается неинициализированной, и поэтому к ней невозможно обратиться никаким способом. Только команды, оставленные компилятором в точке исходного объявления, могут выполнить эту инициализацию. После этого момента TDZ завершается, а переменная может свободно использоваться в оставшейся части области видимости.

Формально у `var` тоже есть область TDZ, но она имеет нулевую длину, а следовательно, остается ненаблюдаемой для наших программ! Наблюдаемые TDZ существуют только у `let` и `const`.

Кстати говоря, определение «временное» (Temporal) в TDZ действительно относится ко времени, а не к позиции в коде. Пример:

```
askQuestion();
// ReferenceError

let studentName = "Suzy";

function askQuestion() {
    console.log(`${ studentName }, do you know?`);
}
```

И хотя позиционно команда `console.log(..)` со ссылкой на `studentName` следует после объявления `let studentName`, по времени функция `askQuestion()` вызывается до обнаружения команды `let`, пока `studentName` все еще находится в TDZ! Отсюда и ошибка.

Существует распространенное заблуждение, будто наличие TDZ говорит о том, что `let` и `const` не поднимаются. Это утверждение неточно или по крайней мере в какой-то степени ошибочно. Поднятие к ним определенно применяется.

Реальное отличие заключается в том, что объявления `let`/`const` не инициализируются автоматически в начале области видимости так, как это происходит с `var`. Тогда возникает *вопрос*, является ли автоматическая инициализация *частью* поднятия или нет? Я думаю, что автоматическая регистрация переменной в начале области видимости (т. е. то, что я называю поднятием) и автоматическая инициализация в начале области видимости (значением `undefined`) являются разными операциями и их не следует сваливать вместе под общим термином «поднятие».

Мы уже видели, что `let` и `const` не выполняют автоматической инициализации в начале области видимости. Но давайте докажем, что `let` и `const` *выполняют* поднятие (автоматическую регистрацию в начале области видимости) при помощи нашего знакомого — замещения (см. «Затенение», глава 3):

```
var studentName = "Kyle";
{
    console.log(studentName);
    // ???
```

```
    // ..

    let studentName = "Suzy";

    console.log(studentName);
    // Suzy
}
```

Что произойдет с первой командой `console.log(..)`? Если `let studentName` не поднимается в начало области видимости, первый вызов `console.log(..)` *должен* вывести `"Kyle"`, верно? Казалось бы, в этот момент есть только внешняя переменная `studentName`, поэтому именно к этой переменной должна обратиться и вывести команда `console.log(..)`.

Но вместо этого первая команда `console.log(..)` выдает ошибку TDZ, потому что переменная `studentName` из внутренней области видимости была поднята (автоматически зарегистрирована в начале области видимости). Однако на этот момент (еще!) не была выполнена автоматическая инициализация внутренней переменной `studentName`; на этот момент она остается неинициализированной, отсюда и нарушение TDZ!

Подведем итог: ошибка TDZ возникает из-за того, что объявления `let`/`const` поднимаются в начало своих областей видимости, но, в отличие от `var`, они откладывают автоматическую инициализацию своих переменных до той точки последовательности выполнения, в которой появляется исходное объявление. Это временное окно, независимо от его длины, и образует область TDZ.

Как избежать ошибок TDZ?

Мой совет: всегда размещайте объявления `let` и `const` в начале области видимости. Уменьшите окно TDZ до нулевой (или почти нулевой) длины, и проблема исчезнет сама собой.

Но почему тема TDZ вообще актуальна? Почему TC39 не потребует, чтобы `let`/`const` автоматически инициализировались так, как это делает `var`? Подождите немного, мы вернемся к ответам на вопросы «почему?» о TDZ в приложении A.

# После инициализации

При работе с переменными возникает гораздо больше нюансов, чем кажется на первый взгляд. *Поднятие*, (*повторное*) *объявление* и *TDZ* часто сбивают с толку разработчиков, особенно перешедших на JS с других языков.

Прежде чем двигаться дальше, убедитесь в том, что в вашей внутренней модели хорошо проработаны эти аспекты областей видимости и переменных JS.

Поднятие обычно описывается как реальный механизм движка JS, но в действительности это скорее метафора для описания различных аспектов обработки объявлений переменных JS в фазе компиляции. Но даже как метафора поднятие предоставляет полезную структуру для рассмотрения жизненного цикла переменной — когда она создается, когда становится доступной для использования и когда перестает существовать.

Первичное и повторное объявление переменных обычно создают путаницу, когда они рассматриваются как операции времени выполнения. Но если рассматривать эти операции в контексте времени компиляции, все странности исчезают.

Ошибка TDZ (Temporal Dead Zone) раздражает. К счастью, ее можно элементарно избежать, если вы всегда будете размещать объявления `let/const` в начале любой области видимости.

В этой главе вы начали успешно осваивать все хитросплетения области видимости переменной. В следующей главе будут представлены факторы, влияющие на наши решения о размещении объявлений в разных областях видимости, и прежде всего во вложенных блоках.

# 6 Ограничение раскрытия областей видимости

До сих пор наше внимание было направлено на объяснение механики того, как работают области видимости и переменные. Теперь, когда мы усвоили эту тему, можно подняться на более высокий уровень абстракции: решения и паттерны, применяемые в программах в целом.

Для начала посмотрим, как и почему стоит использовать разные уровни областей видимости (функции и блоки) для организации переменных программы и конкретно для сокращения чрезмерного раскрытия областей видимости.

## Принцип наименьшего раскрытия

Вполне логично, что функции определяют собственные области видимости. Но почему блоки тоже должны создавать области видимости?

В программировании сформулирован фундаментальный принцип, который обычно применяется в области безопасности ПО, — принцип наименьших привилегий[1] (POLP, Principle of Least Privilege). А разновидность этого принципа, применяемая к на-

[1] https://ru.wikipedia.org/wiki/Принцип_минимальных_привилегий

шему текущему обсуждению, обычно называется принципом наименьшего раскрытия (POLE, Principle of Least Exposure).

Принцип наименьших привилегий выражает защитную позицию программной архитектуры: компоненты системы должны проектироваться так, чтобы они функционировали с наименьшими привилегиями, наименьшим уровнем доступа и наименьшим раскрытием. Если каждый компонент соединяется с другими с минимальными необходимыми возможностями, система в целом становится сильнее с точки зрения безопасности, потому что сбой или взлом одного компонента минимально влияет на остальные части системы.

Если принцип наименьших привилегий направлен на проектирование компонентов на уровне системы, принцип наименьшего раскрытия сосредоточен на более низком уровне; мы применим его к взаимодействиям между областями видимости.

Если следовать принципу наименьшего раскрытия, что именно должно раскрываться в наименьшей степени? Очень просто: переменные, зарегистрированные в каждой области видимости. На это можно взглянуть так: почему не стоит размещать все переменные программы в глобальной области видимости? Наверное, вы и так чувствуете, что это неудачная идея, но стоит разобраться почему. Когда переменные, используемые одной частью программы, раскрываются для другой части через область видимости, часто возникают три главных фактора риска.

- **Конфликты имен:** если вы используете стандартное содержательное имя переменной/функции в двух разных частях программы, но идентификатор происходит из одной общей области видимости (например, глобальной), возникает конфликт имен и очень вероятно, что использование одной частью переменной/функции способом, неожиданным для другой части, приведет к ошибке.

  Например, представьте, что все ваши циклы используют одну глобальную переменную цикла i. Цикл из одной функции вы-

полняется во время итерации цикла из другой функции, и общей переменной i присваивается неожиданное значение.

- **Неожиданное поведение:** если вы раскроете переменные/функции, использование которых должно быть *приватным* для некоторой части программы, это позволит другим разработчикам использовать их непредвиденным образом. Это приведет к нарушению ожидаемого поведения и появлению ошибок.

  Например, если ваша часть программы предполагает, что массив содержит только числа, но чей-то посторонний код обратится к массиву и включит в него логические значения и строки, это может привести к непредвиденным сбоям в вашем коде.

  Что еще хуже, раскрытие *приватных* подробностей подталкивает злоумышленника к попытке обойти ваши ограничения и использовать вашу программу запрещенным способом.

- **Непреднамеренная зависимость:** раскрывая переменные/функции без необходимости, вы подталкиваете других разработчиков к тому, чтобы использовать эти в остальном *приватные* части и зависеть от них. И хотя сейчас работоспособность вашей программы от этого не пострадает, в будущем вы уже не сможете легко провести рефакторинг без риска нарушить работоспособность других частей продукта, которые не контролируете.

  Например, если ваш код зависит от числового массива, а позднее вы решите, что вместо массива лучше использовать другую структуру данных, то тем самым принимаете на себя ответственность за соответствующее изменение других частей продукта.

Применительно к областям видимости переменных/функций принцип наименьшего раскрытия фактически говорит: по умолчанию раскрытие должно ограничиваться абсолютным минимумом, а все остальное должно остаться приватным настолько,

насколько это возможно. Объявляйте переменные в областях видимости наименьшего размера и наибольшего уровня вложенности, вместо того чтобы размещать все в глобальной области видимости (или даже внешней функции).

Если вы спроектируете свою программу соответствующим образом, у вас будет намного больше возможностей для предотвращения этих трех рисков (или, по крайней мере, сведения их к минимуму).

Пример:

```
function diff(x,y) {
    if (x > y) {
        let tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}

diff(3,7); // 4
diff(7,5); // 2
```

В этой функции `diff(..)` необходимо убедиться в том, что значение `y` больше либо равно `x`, чтобы при вычитании (`y - x`) результат был неотрицательным. Если значение `x` изначально больше (т. е. результат будет отрицательным), `x` и `y` меняются местами с использованием переменной `tmp`, чтобы результат оставался положительным.

В этом простом примере вроде бы неважно, находится ли `tmp` внутри блока `if` или же существует на уровне функции, — конечно, эта переменная не должна быть глобальной! Однако в соответствии с принципом наименьшего раскрытия переменная `tmp` должна иметь настолько глубокую область видимости, насколько это возможно. По этой причине мы назначаем `tmp` блоковую область видимости (при помощи `let`) в границах блока `if`.

## Сокрытие в функциональной области видимости

К этому моменту вам должно быть понятно, почему так важно скрывать объявления переменных и функций на самом нижнем уровне (с наибольшим уровнем вложенности) из всех возможных. Но как это сделать?

Вам уже знакомы ключевые слова `let` и `const`, используемые для объявления с блоковой областью видимости; мы еще вернемся к ним для более подробного рассмотрения. Но для начала — как насчет сокрытия объявлений `var` или `function` в областях видимости? Это легко можно сделать, упаковав объявление в область видимости `function`.

Рассмотрим пример, в котором область видимости в границах функции может принести пользу.

Математическая операция «факториал» (записывается в виде `6!`) вычисляет произведение всех целых чисел от заданного вниз до 1 — на самом деле можно остановиться на 2, потому что при умножении на 1 ничего не меняется. Иначе говоря, `6!` — то же самое, что `6 * 5!`, а это то же самое, что `6 * 5 * 4!` и т. д. Из-за природы вычислений, если вы вычислили факториал некоторого числа (например, `4!`), проделывать эту работу заново уже не нужно, так как ответ всегда будет одним и тем же.

Итак, если вы вычисляете факториал 6, а потом вдруг потребуется вычислить факториал 7, это может привести к избыточному вычислению факториалов всех целых чисел от 2 до 6. Но если вы желаете потратить немного памяти ради повышения скорости, проблему избыточных вычислений можно решить кэшированием факториала каждого целого числа после его вычисления:

```
var cache = {};

function factorial(x) {
    if (x < 2) return 1;
    if (!(x in cache)) {
        cache[x] = x * factorial(x - 1);
```

```
    }
    return cache[x];
}

factorial(6);
// 720

cache;
// {
// "2": 2,
// "3": 6,
// "4": 24,
// "5": 120,
// "6": 720
// }

factorial(7);
// 5040
```

Все вычисленные факториалы сохраняются в кэше, так что между несколькими вызовами `factorial(..)` результаты предыдущих вычислений остаются. Однако переменная `cache` очевидно является приватной деталью работы `factorial(..)`, а не чем-то таким, что должно раскрываться во внешней области видимости, особенно в глобальной области видимости.

Здесь функция `factorial(..)` является рекурсивной (т. е. вызывающей саму себя), но это сделано только для компактности кода. С нерекурсивной реализацией анализ областей видимости в отношении кэширования был бы точно таким же.

Но проблема чрезмерного раскрытия не сводится к простому сокрытию переменной `cache` внутри `factorial(..)`, как могло бы показаться. Так как переменная `cache` должна пережить несколько вызовов, она должна находиться в области видимости за пределами этой функции. Что же делать?

Нужно определить для `cache` другую область видимости (между внешней/глобальной областью видимости и внутренней областью видимости `factorial(..)`):

```
// внешняя/глобальная область видимости

function hideTheCache() {
     // промежуточная область видимости, в которой скрывается `cache`
     var cache = {};

     return factorial;

     // **********************

     function factorial(x) {
          // внутренняя область видимости
          if (x < 2) return 1;
          if (!(x in cache)) {
               cache[x] = x * factorial(x - 1);
          }
          return cache[x];
     }
}

var factorial = hideTheCache();

factorial(6);
// 720

factorial(7);
// 5040
```

Функция `hideTheCache()` не имеет другой цели, кроме создания для `cache` области видимости, чтобы значение сохранялось между вызовами `factorial(..)`. Но чтобы функция `factorial(..)` имела доступ к `cache`, она должна определяться в той же области видимости. Затем ссылка на функцию возвращается в виде значения из `hideTheCache()` и сохраняется в переменной внешней области видимости, которая тоже называется `factorial`. Теперь при вызове `factorial(..)` (многократном) его долгосрочная переменная `cache` остается скрытой, но доступной только для `factorial(..)`.

Хорошо, но... будет крайне утомительно определять (и присваивать имя) область видимости функции `hideTheCache(..)` каждый раз, когда возникнет необходимость в сокрытии переменной/функции, особенно если учесть, что мы захотим избежать конфликтов имен и будем присваивать каждому вхождению уникальное имя.

Такой прием — кэширование вычисляемого вывода функции для оптимизации быстродействия, если ожидаются повторные вызовы с теми же входными данными, — весьма распространен в функциональном программировании (FP), где он называется мемоизацией; этот способ кэширования основан на замыканиях (см. главу 7). Также следует учитывать потенциальные проблемы с затратами памяти (см. раздел «Несколько слов о памяти», приложение Б). Библиотеки FP обычно предоставляют оптимизированную и проверенную реализацию мемоизации функций, которые занимают место приведенной функции `hideTheCache(..)`. Мемоизация выходит за рамки нашего обсуждения. За дополнительной информацией обращайтесь к моей книге Functional-Light JavaScript.

Чтобы не определять новую функцию с уникальным именем каждый раз, когда возникнет одна из этих ситуаций «создание области видимости только для сокрытия переменной», возможно, лучше воспользоваться функциональным выражением:

```
var factorial = (function hideTheCache() {
    var cache = {};

    function factorial(x) {
        if (x < 2) return 1;
        if (!(x in cache)) {
            cache[x] = x * factorial(x - 1);
        }
        return cache[x];
    }

    return factorial;
})();

factorial(6);
// 720

factorial(7);
// 5040
```

Но постойте! В этом случае для сокрытия `cache` все равно используется функция для создания области видимости, и в этом случае функция все равно называется `hideTheCache`. И чего мы здесь добились?

Вспомните из раздела «Область видимости имени функции» (глава 3), что происходит с идентификатором имени из функционального выражения. Так как `hideTheCache(..)` определяется как функциональное выражение вместо объявления функции, ее имя находится в отдельной области видимости — по сути, в той же области видимости, что и `cache`, вместо внешней/глобальной области видимости.

Это означает, что всем вхождениям этого функционального выражения можно присвоить одинаковые имена и никаких конфликтов не возникнет. Или, что более уместно, каждому вхождению можно присвоить семантическое имя на основании того, что мы собираемся скрыть, не беспокоясь о том, что выбранное имя может конфликтовать с любой другой областью видимости функционального выражения в программе.

Более того, имя *возможно* полностью опустить — в результате будет получено анонимное функциональное выражение. В приложении А обсуждается важность назначения имен даже для таких функций, создаваемых исключительно ради области видимости.

### Немедленный вызов функциональных выражений

В приведенной рекурсивной программе вычисления факториала есть еще один аспект, который легко упустить: строка в конце функционального выражения с символами `})();`.

Обратите внимание: все функциональное выражение заключено в круглые скобки `(..)`, а затем в конце добавляется вторая пара круглых скобок `()`; она вызывает только что определенное функциональное выражение. Более того, в данном случае первая пара окружающих скобок `(..)` вокруг функционального выражения не является строго необходимой (подробнее об этом чуть позже), но мы используем их ради удобочитаемости.

Итак, мы определяем функциональное выражение, которое немедленно вызывается в программе. У этого распространенного паттерна есть (весьма неожиданное) имя: *немедленно вызываемое*

*функциональное выражение*, или *IIFE* (Immediately Invoked Function Expression).

IIFE полезны в тех ситуациях, когда вы хотите создать область видимости для сокрытия переменных/функций. Так как это выражение, оно может использоваться в любой точке программы JS, в которой допустимо выражение. IIFE могут назначаться имена, как в случае с `hideTheCache()`, или (что бывает намного чаще) они могут оставаться анонимными. Также они могут быть автономными или частью другой команды — `hideTheCache()` возвращает ссылку на функцию `factorial()`, которая затем присваивается переменной `factorial`.

Для сравнения приведу пример автономной IIFE:

```
// внешняя область видимости

(function(){
    // внутренняя область видимости
})();

// снова внешняя область видимости
```

В отличие от предыдущего примера с `hideTheCache()`, где внешние скобки `(..)` были необязательными и включались по стилистическим соображениям, для автономных IIFE они обязательны; с ними функция воспринимается как выражение, а не как команда. Однако ради логической целостности функции IIFE всегда следует заключать в `(..)`.

Технически окружающие скобки `(..)` — не единственный синтаксический способ, гарантирующий, что функция в IIFE будет рассматриваться парсером JS как функциональное выражение. Другие возможности рассматриваются в приложении А.

## Границы функций

Учтите, что использование IIFE для определения области видимости может иметь непредсказуемые последствия в зависимости от окружающего кода. Так как IIFE является полной функцией,

границы функции изменяют поведение некоторых команд/конструкций.

Например, команда `return`, заключенная в IIFE, может изменить свой смысл, потому что после этого `return` будет относиться к функции IIFE. IIFE с нестрелочными функциями также изменяют привязку ключевого слова `this` — об этом подробнее в книге «Объекты и классы». И такие команды, как `break` и `continue`, не будут работать через границу функции IIFE для управления внешним циклом или блоком.

Таким образом, если код, который нужно заключить в область видимости, содержит `return`, `this`, `break` или `continue`, IIFE вряд ли будет лучшим выходом. В этом случае можно рассмотреть возможность создания области видимости с блоком вместо функции.

## Создание областей видимости с блоками

Вероятно, к этому моменту вы уже достаточно уверенно чувствуете себя с созданием областей видимости для ограничения раскрытия идентификаторов. До сих пор мы делали это с использованием области видимости функции (например, IIFE). Но рассмотрим использование объявления `let` с вложенными блоками. В общем случае любая пара фигурных скобок `{ .. }`, являющаяся командой, будет действовать как блок, но не обязательно как область видимости.

Блок становится областью видимости только в случае необходимости для размещения объявлений с блоковой областью видимости (т. е. `let` или `const`). Пример:

```
{
    // не обязательно область видимости (пока)

    // ..

    // теперь мы знаем, что блок должен быть областью видимости
    let thisIsNowAScope = true;
```

```
    for (let i = 0; i < 5; i++) {
        // также является областью видимости, активизируемой
        // после каждой итерации
        if (i % 2 == 0) {
            // просто блок, не область видимости
            console.log(i);
        }
    }
}
// 0 2 4
```

Не все пары фигурных скобок `{ .. }` создают блоки (а следовательно, становятся кандидатами для превращения в области видимости):

- объектные литералы используют пары фигурных скобок `{ .. }` для ограничения своих списков «ключ — значение», но такие объектные значения не являются областями видимости;
- `class` использует фигурные скобки `{ .. }` для определения тела, но блок или область видимости при этом не определяется;
- тело функции заключается в `{ .. }`, но с технической точки зрения это блоком не является — это одна команда для тела функции. Тем не менее это (функциональная) область видимости;
- пара фигурных скобок `{ .. }` в команде `switch` (в которую заключается набор условий `case`) не определяет блок/область видимости.

Кроме этих «неблоковых» примеров пара фигурных скобок `{ .. }` может определить блок, присоединенный к команде (как `if` или `for`), или автономный (как внешняя пара фигурных скобок `{ .. }` в приведенном фрагменте). Явный блок такого типа — если он не содержит объявлений, то не является областью видимости, — не имеет практической цели, хотя и может послужить полезным семантическим сигналом.

Явные автономные блоки `{ .. }` всегда были допустимым элементом синтаксиса JS, но так как до включения `let`/`const` в ES6 они не могли создавать область видимости, то использовались

достаточно редко. После появления ES6 они начинают немного наверстывать упущенное.

В большинстве языков, поддерживающих блоковую область видимости, явная блоковая область видимости стала крайне распространенным паттерном для создания узких сегментов видимости для одной или нескольких переменных. Таким образом, в соответствии с принципом наименьшего раскрытия этот паттерн также должен найти более широкое применение и в JS; используйте (явные) блоковые области видимости для сужения раскрытия идентификаторов до минимального практического уровня.

Явная блоковая область видимости может принести пользу даже внутри другого блока (независимо от того, является внешний блок областью видимости или нет).

Пример:

```
if (somethingHappened) {
    // блок, но не область видимости

    {
        // и блок, и явная область видимости
        let msg = somethingHappened.message();
        notifyOthers(msg);
    }

    // ..

    recoverFromSomething();
}
```

Здесь пара фигурных скобок `{ .. }` в команде `if` создает еще меньшую внутреннюю блоковую область видимости для `msg`, так как эта переменная не нужна для всего блока `if`. Многие разработчики просто ограничили бы видимость `msg` блоком `if` и двинулись бы дальше. И откровенно говоря, если вам приходится просматривать всего несколько строк кода, такое решение вполне можно назвать делом вкуса. Но с ростом кодовой базы проблемы с чрезмерным раскрытием становятся более заметными.

Так ли это важно, чтобы добавлять лишнюю пару скобок `{ .. }` и уровень отступа? Я считаю, что вам стоит следовать принципу наименьшего раскрытия и всегда (в пределах разумного) определять наименьший блок для каждой переменной. Соответственно, я рекомендую использовать дополнительную явную блоковую область видимости, как показано ранее.

Вспомните обсуждение ошибок TDZ из раздела «Неинициализированные переменные (TDZ) (глава 5). Тогда я рекомендовал для минимизации риска ошибок TDZ с объявлениями `let/const` всегда размещать эти объявления в начале области видимости.

Если вдруг в какой-то момент вы разместите объявление `let` в середине области видимости, прежде всего подумайте: «О нет! Опасно — ошибки TDZ!» Если это объявление `let` не задействовано в первой половине блока, используйте явную внутреннюю блоковую область видимости, чтобы дополнительно сузить его раскрытие.

Другой пример с явной блоковой областью видимости:

```
function getNextMonthStart(dateStr) {
    var nextMonth, year;

    {
        let curMonth;
        [ , year, curMonth ] = dateStr.match(
                /(\d{4})-(\d{2})-\d{2}/
            ) || [];
        nextMonth = (Number(curMonth) % 12) + 1;
    }

    if (nextMonth == 1) {
        year++;
    }

    return `${ year }-${
            String(nextMonth).padStart(2,"0")
        }-01`;
}
getNextMonthStart("2019-12-25"); // 2020-01-01
```

Для начала определим области видимости и их идентификаторы:

1. Внешняя/глобальная область видимости содержит один идентификатор — функцию `getNextMonthStart(..)`.
2. Область видимости функции `getNextMonthStart(..)` содержит три идентификатора: `dateStr` (параметр), `nextMonth` и `year`.
3. Пара фигурных скобок `{ .. }` определяет внутреннюю блоковую область видимости, которая включает одну переменную: `curMonth`.

Почему же тогда `curMonth` размещается в явной блоковой области видимости, а не рядом с `nextMonth` и `year` в области видимости функции верхнего уровня? Потому что переменная `curMonth` нужна только для первых двух команд; на уровне области видимости функции она чрезмерно раскрыта.

В этом маленьком примере риски от чрезмерного раскрытия `curMonth` сильно ограничены. Однако преимущества принципа наименьшего раскрытия лучше всего реализуются тогда, когда вы обзаводитесь привычкой минимизировать раскрытие области видимости по умолчанию. Если вы будете последовательно следовать этому принципу даже в маломасштабных ситуациях, это только принесет пользу с ростом вашей программы.

Теперь рассмотрим более содержательный пример:

```
function sortNamesByLength(names) {
    var buckets = [];

    for (let firstName of names) {
        if (buckets[firstName.length] == null) {
            buckets[firstName.length] = [];
        }
        buckets[firstName.length].push(firstName);
    }

    // блок для сужения области видимости
    {
        let sortedNames = [];

        for (let bucket of buckets) {
```

```
            if (bucket) {
                // каждый массив сортируется по алфавиту
                bucket.sort();

                // присоединить отсортированные имена
                // к текущему списку
                sortedNames = [
                    ...sortedNames,
                    ...bucket
                ];
            }
        }

        return sortedNames;
    }
}

sortNamesByLength([
    "Sally",
    "Suzy",
    "Frank",
    "John",
    "Jennifer",
    "Scott"
]);
// [ "John", "Suzy", "Frank", "Sally",
// "Scott", "Jennifer" ]
```

Здесь шесть идентификаторов объявляются в пяти разных областях видимости. Могли бы все эти переменные существовать в одной внешней/глобальной области видимости? Технически да, потому что они имеют уникальные имена, что исключает возможные конфликты имен. Но такая организация кода была бы крайне примитивной и с большой вероятностью привела бы как к недоразумениям, так и к будущим ошибкам.

Мы выделяем каждую из этих переменных во внутреннюю вложенную область видимости. Каждая переменная определяется в области видимости с наибольшим уровнем вложенности, чтобы программа работала так, как требуется.

Переменную `sortedNames` можно было бы определить в области видимости функции верхнего уровня, но она используется только

во второй половине функции. Чтобы избежать чрезмерного раскрытия этой переменной в области видимости более высокого уровня, мы снова следуем принципу наименьшего раскрытия и оформляем ее с блоковой видимостью во внутренней явной блоковой области видимости.

### var и let

Теперь поговорим об объявлении `var buckets`. Эта переменная используется во всей функции (кроме последней команды `return`). Любая переменная, которая должна быть доступна во всем коде функции (или почти во всем), должна объявляться так, чтобы область ее использования была очевидной.

Параметр `names` не используется во всей функции, но ограничить область видимости параметра невозможно, поэтому он ведет себя как объявление с областью видимости функции.

Почему же мы используем `var` вместо `let` для объявления переменной `buckets`? Для выбора `var` есть как семантические, так и технические причины.

С точки зрения стилистики объявление `var` всегда, с первых дней JS, означало переменную, принадлежащую всей функции. Как было указано в разделе «Лексическая видимость» (глава 1), `var` присоединяется к ближайшей вмещающей области видимости функции, где бы она ни находилась. Это справедливо даже в том случае, если `var` размещается внутри блока:

```
function diff(x,y) {
    if (x > y) {
        var tmp = x; // `tmp` is function-scoped
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Даже при том что объявление `var` располагается внутри блока, его область видимости определяется функцией (`diff(..)`), а не блоком.

И хотя объявление `var` может располагаться в блоке (и при этом все равно будет иметь функциональную область видимости), я не рекомендую использовать этот прием, кроме особых случаев (рассмотренных в приложении А). В остальных случаях использование `var` следует ограничивать областью верхнего уровня функции.

Почему бы не использовать `let` в этой же точке? Потому что `var` визуально отличается от `let` и потому четко сигнализирует: эта переменная имеет функциональную область видимости. Использование `let` в области видимости верхнего уровня, особенно если оно не находится в нескольких начальных строках функции и когда все остальные объявления в блоках используют `let`, не привлекает внимания к отличиям от объявлений с функциональной областью видимости.

Иначе говоря, я считаю, что `var` лучше передает функциональную область видимости, чем `let`, а `let` одновременно передает (и реализует) блоковую область видимости там, где недостаточно `var`. Пока вашим программам понадобятся переменные как с функциональной, так и с блоковой областью видимости, самое разумное и удобочитаемое решение — использовать как объявления `var`, *так и* `let`, каждое для наиболее подходящей цели.

Существуют и другие семантические и практические причины для выбора `var` или `let` в разных ситуациях. Ситуации, в которых уместно применение `var` и `let`, более подробно рассматриваются в приложении А.

Моя рекомендация использовать как `var`, так и `let` очевидным образом спорна и противоречит мнению большинства. Гораздо чаще встречаются утверждения вида: «Объявления `var` сломаны, `let` их чинит» или: «Никогда не используйте `var`, `let` — идеальная замена». Эти мнения имеют право на существование, но это всего лишь мнения, как и мое. Объявления var не сломаны и не устарели; они работали с первых дней существования JS и будут работать, пока существует JS.

## Где использовать let?

Мой совет ограничить использование `let` (почти всегда) только функциональной областью видимости верхнего уровня означает, что в большинстве других объявлений должно использоваться ключевое слово `let`. Но возможно, вас все еще интересует, как решить, какой тип выбрать для каждого объявления в вашей программе?

Принцип наименьшего раскрытия уже направляет такие решения, но давайте выразим критерий выбора явно. Выбор не зависит от того, какое ключевое слово положено использовать в конкретной ситуации. Чтобы принять решение, спросите себя, какое минимальное раскрытие области видимости будет достаточным для этой переменной.

Ответив на этот вопрос, вы будете знать, к какой области видимости должна принадлежать переменная — блоковой или функциональной. Если вы изначально решили, что переменная должна иметь блоковую область видимости, а позднее осознаете, что ее следует поднять до функциональной области видимости, это повлияет не только на местоположение объявления этой переменной, но и на используемое при объявлении ключевое слово. Процесс принятия решений должен проходить именно так.

Если объявление принадлежит блоковой области видимости, используйте `let`. Если оно принадлежит функциональной области видимости, используйте `var` (еще раз: это только мое мнение).

Чтобы понять суть этого решения, можно подумать, как бы выглядела версия этой программы до ES6. Например, вспомним приведенную выше функцию `diff(..)`:

```
function diff(x,y) {
    var tmp;

    if (x > y) {
        tmp = x;
        x = y;
        y = tmp;
    }
```

```
    return y - x;
}
```

В этой версии `diff(..)` переменная `tmp` очевидно объявляется в функциональной области видимости. Подходит ли это для `tmp`? На мой взгляд, нет. Переменная `tmp` нужна только для этих нескольких команд. Для команды `return` она не нужна, поэтому должна иметь блоковую область видимости.

До выхода ES6 ключевого слова `let` не было, поэтому придать ему блоковую область видимости было невозможно. Но для передачи ваших намерений можно было использовать доступные средства:

```
function diff(x,y) {
    if (x > y) {
        // `tmp` по-прежнему имеет функциональную область
        // видимости,но ее размещение здесь является
        // семантическим сигналом о блоковой области видимости
        var tmp = x;
        x = y;
        y = tmp;
    }

    return y - x;
}
```

Объявление `var` для переменной `tmp` внутри команды `if` сигнализирует читателю кода, что `tmp` принадлежит этому блоку. Несмотря на то что JS не ограничивает область видимости, семантический сигнал все равно принесет некоторую пользу для читателя вашего кода.

Следуя этому принципу, вы можете найти все объявления `var`, расположенные внутри подобных блоков, и переключить их на `let` для передачи семантического сигнала. На мой взгляд, это правильный способ использования `let`.

В другом примере исторически использовалось объявление `var`, но теперь практически всегда в цикле `for` должно использоваться `let`:

```
for (var i = 0; i < 5; i++) {
    // ...
}
```

Где бы ни определялся такой цикл, переменная `i`, по сути, всегда используется только внутри цикла; в этом случае принцип наименьшего раскрытия требует, чтобы она объявлялась с ключевым словом `let` вместо `var`:

```
for (let i = 0; i < 5; i++) {
    // ...
}
```

Подобное переключение с `var` на `let` нарушит работоспособность вашего кода только в одном случае: если он зависит от обращения к переменной цикла (`i`) за пределами/после цикла:

```
for (var i = 0; i < 5; i++) {
    if (checkValue(i)) {
        break;
    }
}

if (i < 5) {
    console.log("The loop stopped early!");
}
```

Этот паттерн встречается не так уж редко, но по мнению большинства разработчиков, он отдает плохо структурированным кодом. В подобных случаях рекомендуется использовать для этой цели другую переменную с внешней областью видимости:

```
var lastI;

for (let i = 0; i < 5; i++) {
    lastI = i;
    if (checkValue(i)) {
        break;
    }
}

if (lastI < 5) {
    console.log("The loop stopped early!");
}
```

Переменная `lastI` нужна во всей области видимости, поэтому она объявляется с ключевым словом `var`. Переменная `i` нужна только

в (каждой) итерации цикла, поэтому она объявляется с ключевым словом `let`.

### В чем загвоздка?

До настоящего момента утверждалось, что `var` и параметры имеют функциональную область видимости, а `let`/`const` сигнализируют об объявлениях с блоковой областью видимости. Существует только одно маленькое исключение, заслуживающее упоминания: секция `catch`.

С момента появления `try..catch` в ES3 (в 1999 году) в секции `catch` использовалась дополнительная (малоизвестная) возможность объявления блоковой области видимости:

```
try {
    doesntExist();
}
catch (err) {
    console.log(err);
    // ReferenceError: 'doesntExist' is not defined
    // ^^^^ сообщение, выводимое для перехваченного исключения

    let onlyHere = true;
    var outerVariable = true;
}

console.log(outerVariable); // true

console.log(err);
// ReferenceError: 'err' is not defined
// ^^^^ другое (неперехваченное) исключение
```

Переменная `err`, объявленная в секции `catch`, имеет блоковую область видимости для данного блока. Блок секции `catch` может содержать другие объявления с блоковой областью видимости, создаваемые `let`. Однако объявление `var` внутри этого блока все еще присоединяется к внешней функциональной/глобальной области видимости.

В ES2019 (недавно, на момент написания книги) секции `catch` были изменены и их объявление стало необязательным; если объявление опущено, то блок `catch` (по умолчанию) уже не является областью видимости, но при этом он остается блоком!

Таким образом, если нужно отреагировать на *факт возникновения исключения* (после чего можно корректно продолжить работу), но само значение ошибки вас не интересует, объявление `catch` можно опустить:

```
try {
    doOptionOne();
}
catch { // Объявление catch опущено
    doOptionTwoInstead();
}
```

Это небольшое, но приятное упрощение синтаксиса для довольно распространенного случая использования; оно также может быть чуть более эффективным для удаления избыточных областей видимости!

## Объявления функций в блоках (FiB)

Итак, вы видели, что объявления с `let` или `const` имеют блоковую область видимости, а объявления `var` — функциональную. А как насчет объявлений, размещаемых непосредственно внутри блоков? Эта возможность называется *FiB* (Functions in Blocks).

Обычно мы рассматриваем объявления функций как эквиваленты объявлений `var`. Так значит, они имеют функциональную область видимости, как и `var`?

Нет и да. Знаю, это звучит странно. Разберем подробнее:

```
if (false) {
    function ask() {
        console.log("Does this run?");
    }
}
ask();
```

Как вы думаете, что сделает эта программа? Три возможных варианта:

1. При вызове `ask()` может произойти исключение `ReferenceError`, потому что идентификатор `ask` имеет блоковую область видимости для блока `if`, а следовательно, недоступен во внешней/глобальной области видимости.
2. При вызове `ask()` может произойти исключение `TypeError`, потому что идентификатор `ask` существует, но он содержит `undefined` (потому что команда `if` не выполняется), а следовательно, не является вызываемой функцией.
3. Вызов `ask()` выполняется правильно и выводит сообщение Does it run?

А теперь самая загадочная часть: в зависимости от того, в какой среде JS будет выполняться этот фрагмент кода, вы можете получить разные результаты! Это одна из немногих странных областей, в которых существующее унаследованное поведение противоречит предсказуемости результата.

Спецификация JS гласит, что объявления функций внутри блоков имеют блоковую область видимости, поэтому ответом должен быть пункт (1). Однако большинство браузерных ядер JS (включая движок v8, который происходит от Chrome, но также используется в Node) ведет себя в соответствии с пунктом (2); это означает, что идентификатор имеет область видимости вне блока `if`, но значение-функция не инициализируется автоматически, поэтому оно остается равным `undefined`.

Почему браузерным движкам JS разрешается нарушать своим поведением спецификацию? Потому что эти движки уже обладали поведением, связанным с FiB, до появления блоковой видимости в ES6, и существовали опасения, что изменения, направленные на соответствие спецификации, могут нарушить работоспособность существующего кода JS веб-сайтов. Из-за этого в приложении B спецификации JS было сделано исключение, позволяющее некоторые отклонения для браузерных движков JS (и только!).

Обычно Node не относится к браузерным средам JS, так как обычно работает на сервере. Однако движок Node v8 является общим с браузером Chrome (и Edge). Так как движок v8 сначала был браузерным движком JS, он включает исключение из приложения B, а это означает, что браузерные исключения распространяются на Node.

Одним из самых распространенных сценариев использования для размещения объявления функции в блоке является условное определение функции тем или иным способом (например, в команде `if..else`) в зависимости от некоторого состояния среды. Пример:

```
if (typeof Array.isArray != "undefined") {
    function isArray(a) {
        return Array.isArray(a);
    }
}
else {
    function isArray(a) {
        return Object.prototype.toString.call(a)
            == "[object Array]";
    }
}
```

Такое структурирование кода по соображениям эффективности выглядит соблазнительно, так как проверка `typeof Array.isArray` выполняется только один раз, в отличие от определения всего одной версии `isArray(..)` и размещения команды `if` внутри нее — в этом случае каждый вызов будет сопровождаться избыточной проверкой.

Кроме рисков, связанных с расхождениями FiB, у условного определения функций есть и другая проблема: оно усложняет отладку таких программ. Если вы столкнетесь с ошибкой в функции `isArray(..)`, вам придется сначала вычислить, какая именно реализация `isArray(..)` при этом выполнялась! А иногда ошибка может возникнуть из-за того, что была выбрана неправильная реализация из-за ошибки в условии! Если вы определили несколько версий функции, такую программу всегда труднее понять и она всегда создает больше проблем с сопровождением.

В дополнение к предыдущим фрагментам, с FiB также связан ряд других граничных случаев; скорее всего, такое поведение в разных браузерных и небраузерных средах JS (движках JS, которые не базируются на браузерах) будет с большой вероятностью изменяться. Пример:

```
if (true) {
    function ask() {
        console.log("Am I called?");
    }
}

if (true) {
    function ask() {
        console.log("Or what about me?");
    }
}

for (let i = 0; i < 5; i++) {
    function ask() {
        console.log("Or is it one of these?");
    }
}

ask();

function ask() {
    console.log("Wait, maybe, it's this one?");
}
```

Напомню, что поднятие функции в соответствии с описанием «Когда можно использовать переменную?» (глава 5) может навести на мысль, что последний вызов `ask()` из этого фрагмента с сообщением Wait, maybe... поднимется над вызовом `ask()`. Так как это последнее объявление функции с таким именем, оно должно «победить», верно? К сожалению, нет.

Не буду даже пытаться документировать все странные граничные случаи или пытаться объяснить, почему каждый из них ведет себя именно так, а не иначе. На мой взгляд, такая информация нужна только для знатоков экзотических нюансов унаследованного поведения.

Когда я говорю о FiB, меня интересует другое: какой совет я могу дать, чтобы обеспечить предсказуемую работу вашего кода во всех обстоятельствах?

На мой взгляд, единственный практичный способ избежать капризов FiB — просто полностью избегать FiB. Другими словами, никогда не размещайте объявления функций непосредственно внутри любого блока. Всегда размещайте объявления функций в любой точке области верхнего уровня функции (или глобальной области видимости).

Таким образом, в более раннем примере `if..else` я бы порекомендовал избегать условного определения функций, если это возможно. Да, может быть, такое решение будет чуть менее производительным, но в целом это лучшее решение:

```
function isArray(a) {
    if (typeof Array.isArray != "undefined") {
        return Array.isArray(a);
    }
    else {
        return Object.prototype.toString.call(a)
            == "[object Array]";
    }
}
```

Если снижение быстродействия создает критические проблемы для вашего приложения, то рекомендую рассмотреть следующий подход:

```
var isArray = function isArray(a) {
    return Array.isArray(a);
};

// переопределите определение, если это необходимо
if (typeof Array.isArray == "undefined") {
    isArray = function isArray(a) {
        return Object.prototype.toString.call(a)
            == "[object Array]";
    };
}
```

Важно заметить, что здесь в команде `if` размещается функциональное выражение, а не объявление. Размещение функциональных выражений в блоках — абсолютно нормальное и допустимое решение. В нашем обсуждении FiB речь идет о нежелательности функциональных объявлений в блоках.

Даже если вы протестировали свою программу и она работает правильно, мелкие преимущества от использования FiB в вашем коде намного меньше будущих потенциальных рисков путаницы от других разработчиков или отклонений при выполнении вашего кода в других средах JS.

FiB не стоит того, и от этой возможности стоит держаться подальше.

## Напоследок о блоках

Правила лексических областей видимости в языках программирования существуют для того, чтобы правильно организовать переменные в вашей программе — как для практических целей, так и для передачи семантических сигналов о коде.

И один из самых важных организационных приемов — предотвращение раскрытия переменных в нежелательных областях видимости (принцип наименьшего раскрытия). Хочется верить, что вы теперь понимаете блоковую область видимости намного лучше, чем прежде.

Надеюсь, вы чувствуете, что ваша основа для понимания лексической видимости заметно укрепилась. И с этой прочной основы можно перейти к непростой теме следующей главы — замыканиям.

# 7 Использование замыканий

Пока что мы подробно изучали все тонкости лексических областей видимости и их влияния на организацию и использование переменных в наших программах.

Сейчас мы вернемся на более общий уровень абстракции и переключимся на традиционно пугающую тему замыканий. Не бойтесь! Разобраться в ней можно и без докторской степени в области компьютерной теории. Наша общая цель в этой книге — не просто разобраться в областях видимости, а более эффективно использовать их в структуре программ; без замыканий сделать это не получится.

Вспомните главный вывод из главы 6: принцип наименьшего раскрытия (POLE) рекомендует использовать блоковые (и функциональные) области видимости для ограничения раскрытия переменных в областях видимости. Соблюдение этого принципа делает ваш код более понятным и простым в сопровождении и помогает избежать многих потенциальных ловушек (конфликтов имен и т. д.).

Замыкания в каком-то смысле расширяют этот принцип: если переменные понадобятся позже, то вместо того чтобы размещать их в больших внешних областях видимости, мы можем инкапсулировать их (сузить их область видимости), но при этом сохранить

их доступность из функций, чтобы расширить возможности их использования. Функции *сохраняют* информацию об этих переменных с ограниченной областью видимости в замыканиях.

Пример замыканий уже встречался в предыдущей главе (`factorial(..)` в главе 6), и вы почти наверняка пользовались ими в своих программах. Если вам когда-либо доводилось писать функцию обратного вызова, которая обращалась к переменным за пределами своей области видимости... да, представьте — это было замыкание.

Замыкание — одна из самых важных характеристик языка, изобретенных в программировании. Замыкания лежат в основе многих фундаментальных парадигм программирования, включая функциональное программирование (FP), модули и даже в какой-то степени объектно-ориентированное проектирование. Хорошо владеть JS и эффективно применять многие важные паттерны проектирования без понимания замыканий не выйдет.

Для рассмотрения всех аспектов замыканий в этой главе пришлось дать много продолжительных объяснений и примеров кода. Не торопитесь и убедитесь в том, что вы полностью поняли каждую часть, прежде чем переходить к следующей.

## Как увидеть замыкание

Изначально замыкание было математической концепцией из области лямбда-исчисления. Но я не собираюсь сыпать математическими формулами или использовать заумные термины и обозначения для его определения.

Вместо этого я намерен сосредоточиться на практической точке зрения. Мы начнем с определения замыканий в контексте того, что можно наблюдать в различных аспектах поведения наших программ, в отличие от того, что происходило бы, если бы замыкания в JS не поддерживались. Но позднее в этой главе мы рассмотрим замыкания с другой точки зрения.

Замыкание является аспектом поведения функций и *только* функций. Если вы работаете не с функцией, то замыкание не действует. Объект не может иметь замыкания, класс не может иметь замыкания (хотя его отдельные функции/методы — могут). Короче, замыкания присущи только функциям.

Чтобы замыкание проявилось, функция должна быть вызвана, причем вызвана не в той ветви цепочки областей видимости, в которой она была определена. Функция, выполняемая в той же области видимости, в которой она была определена, будет работать одинаково независимо от того, возможны замыкания или нет; с позиций наблюдаемости и определения это замыканием не является.

Рассмотрим пример кода с помеченными цветными областями видимости (из главы 2):

```
// внешняя/глобальная область видимости: КРАСНЫЙ(1)

function lookupStudent(studentID) {
    // области видимости функции: СИНИЙ(2)

    var students = [
        { id: 14, name: "Kyle" },
        { id: 73, name: "Suzy" },
        { id: 112, name: "Frank" },
        { id: 6, name: "Sarah" }
    ];

    return function greetStudent(greeting){
        // область видимости функции: ЗЕЛЕНЫЙ(3)

        var student = students.find(
            student => student.id == studentID
        );

        return `${ greeting }, ${ student.name }!`;
    };
}

var chosenStudents = [
    lookupStudent(6),
    lookupStudent(112)
];
```

```
// обращение к свойству name функции:
chosenStudents[0].name;
// greetStudent

chosenStudents[0]("Hello");
// Hello, Sarah!

chosenStudents[1]("Howdy");
// Howdy, Frank!
```

Первое, что следует заметить в этом коде, — что внешняя функция `lookupStudent(..)` создает и возвращает внутреннюю функцию с именем `greetStudent(..)`. `lookupStudent(..)` вызывается дважды, создавая два разных экземпляра своей внутренней функции `greetStudent(..)`; оба экземпляра сохраняются в массиве `chosenStudents`.

Чтобы убедиться в этом, мы проверяем свойство `.name` возвращенной функции, хранящейся в `chosenStudents[0]`, и это в самом деле оказывается экземпляром внутренней функции `greetStudent(..)`.

После завершения каждого вызова `lookupStudent(..)` кажется, что все внутренние переменные пропадают, а их память освобождается в ходе сборки мусора (GC). Внутренняя функция — единственное, что возвращается и сохраняется. Но здесь-то в поведении возникают различия, которые мы можем наблюдать.

Хотя функция `greetStudent(..)` получает один аргумент в параметре с именем `greeting`, она также обращается к `students` и `studentID` — идентификаторам, происходящим из окружающей области видимости `lookupStudent(..)`. Каждая из ссылок из внутренней функции на переменную во внешней области видимости называется замыканием. В научной терминологии каждый экземпляр `greetStudent(..)` *замыкается* по внешним переменным `students` и `studentID`. Что же делают замыкания в конкретном, наблюдаемом смысле?

Замыкание позволяет `greetStudent(..)` продолжать обращаться к этим внешним переменным даже после завершения внешней области видимости (после завершения каждого вызова `lookupStudent(..)`). Вместо того чтобы уничтожаться в ходе сборки мусора,

экземпляры `students` и `studentID` будут оставаться в памяти. И позднее, при вызове экземпляра `greetStudent(..)`, эти переменные все еще будут доступны с сохранением своих текущих значений.

Если бы функции JS не имели замыканий, то завершение каждого вызова `lookupStudent(..)` немедленно уничтожало бы свою область видимости, а переменные `students` и `studentID` уничтожались бы в ходе сборки мусора. Что произойдет, когда позднее мы вызываем одну из функций `greetStudent(..)`?

Если функция `greetStudent(..)` попытается обратиться к тому, что, по ее мнению, является переменной из области видимости СИНИЙ (2), но эта переменная не существует (к настоящему моменту), разумно предположить, что мы получим ошибку `ReferenceError`, верно?

Однако ошибки не будет. Тот факт, что выполнение `chosenStudents[0]("Hello")` работает и возвращает сообщение Hello, Sarah!, означает, что функция все еще может обращаться к переменным `students` и `studentID` — непосредственно наблюдаемый эффект замыкания!

## Замыкание и стрелки

В действительности в предыдущем обсуждении была упущена маленькая подробность, которую, вероятно, упустили многие читатели!

Из-за того, насколько компактен синтаксис стрелочных функций `=>`, легко забыть, что они тоже создают область видимости (см. раздел «Стрелочные функции», глава 3). Стрелочная функция `student => student.id == studentID` создает еще одну область видимости внутри области видимости функции `greetStudent(..)`.

Развивая метафору цветных банок и камешков из главы 2, если бы мы построили цветную диаграмму для этого кода, на ней появилась бы четвертая область видимости на уровне с максимальной вложенностью, поэтому понадобился бы четвертый цвет; например, для этой цели можно выбрать для этой области видимости обозначение ОРАНЖЕВЫЙ(4):

```
var student = students.find(
    student =>
        // область видимости функции: ОРАНЖЕВЫЙ(4)
        student.id == studentID
);
```

Ссылка `studentID` в СИНИЙ (2) в действительности находится внутри области видимости ОРАНЖЕВЫЙ (4) вместо области видимости ЗЕЛЕНЫЙ (3) функции `greetStudent(..)`; кроме того, параметр `student` стрелочной функции относится к области видимости ОРАНЖЕВЫЙ (4), замещая `student` из ЗЕЛЕНЫЙ (3).

Как следствие, стрелочная функция, передаваемая в качестве обратного вызова методу `find(..)` массива, должна содержать замыкание по `studentID`, а не по функции `greetStudent(..)`, содержащей это замыкание. Особых проблем это не создает; все работает, как и ожидалось. Очень важно помнить о том, что даже крошечные стрелочные функции могут участвовать в системе замыканий.

## Накопление замыканий

Рассмотрим один из канонических примеров, часто приводимых для замыканий:

```
function adder(num1) {
    return function addTo(num2){
        return num1 + num2;
    };
}

var add10To = adder(10);
var add42To = adder(42);

add10To(15);  // 25
add42To(9);   // 51
```

Каждый экземпляр внутренней функции `addTo(..)` замыкается по своей собственной переменной `num1` (со значениями 10 и 42 соответственно), так что `num1` не исчезает из-за завершения `adder(..)`. Когда позднее мы вызываем один из этих внутренних

экземпляров `addTo(..)` (например, `add10To(15)`), переменная `num1` из его замыкания продолжает существовать и все еще содержит исходное значение 10. Таким образом, операция может выполнить сложение `10 + 15` и возвращает ответ 25.

В предыдущем абзаце можно легко упустить одну важную подробность, поэтому я хочу снова подчеркнуть ее: замыкание связывается с экземпляром функции вместо его лексического определения. В предыдущем фрагменте существует только одна внутренняя функция `addTo(..)`, определяемая внутри `adder(..)`, поэтому может показаться, что подразумевается только одно замыкание.

Но в действительности при каждом выполнении внешней функции `adder(..)` создается новый экземпляр внутренней функции `addTo(..)` и для каждого нового экземпляра создается новое замыкание. Таким образом, каждый экземпляр внутренней функции (`add10To(..)` и `add42To(..)` в нашей программе) содержит собственное замыкание по своему экземпляру области видимости для этого выполнения `adder(..)`.

И хотя замыкание основано на лексической области видимости, которая обрабатывается во время компиляции, замыкание наблюдается как характеристика экземпляров функций во время выполнения.

### Живая ссылка, а не снимок

В обоих примерах из предыдущих разделов **значение читается из переменной**, находящейся в замыкании. При этом может сложиться впечатление, что замыкание представляет собой «моментальный снимок» значения на некоторый момент. Это весьма распространенное заблуждение.

На самом деле замыкание представляет собой живую ссылку, которая сохраняет доступ к полноценной переменной. Вы не ограничиваетесь простым чтением значения; переменную в замыкании также можно обновлять (присваивать ей новое значение).

Создавая замыкание по переменной в функции, мы можем продолжать пользоваться этой переменной (для чтения и записи), пока ссылка на функцию существует в программе и в любой точке, в которой мы хотим вызвать эту функцию. Вот почему замыкание считается исключительно мощным механизмом, который находит применение во многих областях программирования!

На рис. 4 изображены связи между экземплярами функций и областями видимости.


**Рис. 4.** Наглядное представление замыканий

Как видно из рис. 4, каждый вызов `adder(..)` создает новую область видимости (2), которая содержит переменную `num1`, а также новый экземпляр функции `addTo(..)` как область видимости (3). Обратите внимание на то, что экземпляры функций (`addTo10(..)` и `addTo42(..)`) присутствуют и вызываются из области видимости (1).

Рассмотрим пример с обновлением переменной из замыкания:

```
function makeCounter() {
    var count = 0;

    return getCurrent(){
        count = count + 1;
        return count;
    };
}
```

```
var hits = makeCounter();

// позднее

hits(); // 1

// позднее
hits(); // 2
hits(); // 3
```

Переменная `count` включается в замыкание внутренней функции `getCurrent()`, которая удерживает эту переменную от уничтожения при сборке мусора. Функция `hits()` вызывает `access` *и* обновляет эту переменную, возвращая увеличенный счетчик при каждом вызове.

Хотя окружающая область видимости замыкания обычно происходит от функции, в принципе это не обязательно; необходимым условием является лишь наличие внутренней функции во внешней области видимости:

```
var hits;
{ // внешняя область видимости (но не функция)
    let count = 0;
    hits = function getCurrent(){
        count = count + 1;
        return count;
    };
}
hits();   // 1
hits();   // 2
hits();   // 3
```

Я намеренно определил `getCurrent()` как функциональное выражение, а не как объявление функции. Это связано не с замыканием, а со странными особенностями FiB (глава 6).

Так как очень легко может возникнуть ошибочное впечатление, что замыкания ориентированы на значения, а не на переменные, разработчики нередко попадают в ловушку, когда пытаются использовать замыкание для фиксации значения на некоторый момент. Пример:

```
var studentName = "Frank";

var greeting = function hello() {
    // В замыкании используется `studentName`,
    // а не "Frank"
    console.log(
        `Hello, ${ studentName }!`
    );
}

// позднее
studentName = "Suzy";

// позднее

greeting();
// Hello, Suzy!
```

При определении `greeting()` (т. е. `hello()`), когда `studentName` содержит значение `"Frank"` (перед повторным присваиванием `"Suzy"`), часто ошибочно предполагается, что замыкание сохранит значение `"Frank"`. Но в замыкание `greeting()` включается переменная `studentName`, а не ее значение. При вызове `greeting()` будет использовано текущее значение переменной (`"Suzy"` в данном случае).

Классический пример этой ошибки — определение функций в цикле:

```
var keeps = [];

for (var i = 0; i < 3; i++) {
    keeps[i] = function keepI(){
        // замыкание по `i`
        return i;
    };
}

keeps[0](); // 3 -- ПОЧЕМУ!?
keeps[1](); // 3
keeps[2](); // 3
```

В подобных примерах замыканий обычно используется `setTimeout()` или другой обратный вызов (например, обработчик события) внутри цикла. Я упростил пример, сохранив ссылки на функции в массиве, чтобы нам не приходилось учитывать асинхронность в своем анализе. Принцип замыкания остается неизменным.

Возможно, вы ожидали, что вызов `keeps[0]()` вернет 0, так как функция была создана при первой итерации цикла, когда значение `i` было равно 0. Но и это предположение происходит оттого, что замыкания рассматриваются как ориентированные на значения, а не как ориентированные на переменные.

Структура цикла `for` может создать ложное впечатление, что каждая итерация получает собственную новую переменную `i`; на самом деле в программе существует только одна переменная `i`, потому что она была объявлена с ключевым словом `var`.

Каждая сохраненная функция возвращает 3, потому что к концу цикла единственной переменной `i` в программе было присвоено значение 3. Каждая из трех функций в массиве `keeps` имеет индивидуальные замыкания, но все они замыкаются по одной общей переменной `i`.

Конечно, одна переменная в любой момент может хранить только одно значение. Таким образом, если вы хотите сохранить на будущее несколько значений, понадобится отдельная переменная для каждого.

Как сделать это в приведенном фрагменте цикла? Создадим новую переменную в каждой итерации:

```
var keeps = [];

for (var i = 0; i < 3; i++) {
    // при каждой итерации создается новая переменная `j`,
    // которой присваивается копия значения `i` на данный момент
    let j = i;

    // переменная `i` здесь еще не замкнута, поэтому ничто не
    // мешает непосредственно использовать ее текущее значение
    // при каждой итерации цикла
    keeps[i] = function keepEachJ(){
        // замыкание по `j`, не по `i`!
        return j;
    };
}
keeps[0](); // 0
keeps[1](); // 1
keeps[2](); // 2
```

Каждая функция теперь замыкается по отдельной (новой) переменной из каждой итерации, хотя всем им присвоено имя `j`. И каждой переменной `j` присваивается копия значения `i` на тот момент итерации цикла; значение `j` никогда не изменяется. А значит, теперь все три функции вернут ожидаемые значения: 0, 1 и 2!

И снова следует помнить, что даже если бы в программе использовались асинхронные вызовы (например, передача каждой внутренней функции `keepEachJ()` при вызове `setTimeout(..)` или другой разновидности подписки на обработчики событий), будет наблюдаться то же поведение замыканий.

Вспомните раздел «Циклы» главы 5, который демонстрирует, как объявление `let` в цикле `for` создает не только одну переменную для цикла, но и новую переменную для *каждой итерации* цикла. Этот трюк/странность — именно то, что необходимо для наших замыканий в циклах:

```
var keeps = [];

for (let i = 0; i < 3; i++) {
    // `let i` автоматически создает новую переменную `i`
    // для каждой итерации!
    keeps[i] = function keepEachI(){
        return i;
    };
}

keeps[0](); // 0
keeps[1](); // 1
keeps[2](); // 2
```

Так как мы используем `let`, создаются три переменные `i`, по одной для каждого цикла, и все три замыкания работают именно так, как ожидалось.

## Типичные замыкания: Ajax и события

Замыкания чаще всего встречаются при использовании обратных вызовов:

```
function lookupStudentRecord(studentID) {
    ajax(
        `https://some.api/student/${ studentID }`,
        function onRecord(record) {
            console.log(
                `${ record.name } (${ studentID })`
            );
        }
    );
}

lookupStudentRecord(114);
// Frank (114)
```

Функция обратного вызова `onRecord(..)` будет вызвана в какой-то момент в будущем — после того как вернется ответ от вызова Ajax. Этот вызов произойдет во внутренней реализации служебной функции `ajax(..)`, откуда бы она ни была вызвана. Более того, когда это произойдет, вызов `lookupStudentRecord(..)` уже давно завершится.

Почему же переменная `studentID` все еще существует и остается доступной для обратного вызова? Из-за замыкания.

Обработчики событий — еще одно стандартное применение замыканий:

```
function listenForClicks(btn,label) {
    btn.addEventListener("click",function onClick(){
        console.log(
            `The ${ label } button was clicked!`
        );
    });
}

var submitBtn = document.getElementById("submit-btn");

listenForClicks(submitBtn,"Checkout");
```

Параметр `label` замыкается из обратного вызова обработчика событий `onClick(..)`. При нажатии кнопки `label` все еще существует и может использоваться. И это объясняется замыканием.

## А если я не вижу?

Вероятно, вы слышали этот часто встречающийся философский вопрос: раздается ли в лесу звук падающего дерева, если в лесу никого нет?

На самом деле это глупая философская эквилибристика. Конечно, с научной точки зрения звуковые волны создаются. Но суть в другом: *имеет ли значение*, издается звук или нет?

Вспомните, что в нашем определении особое внимание уделялось наблюдаемости. Если замыкание существует (в техническом, реализационном или академическом смысле), но не может наблюдаться в наших программах, имеет ли это значение? Нет.

Чтобы подчеркнуть этот момент, рассмотрим несколько примеров, *не* основанных на замыканиях с возможностью наблюдения.

Начнем с вызова функции, использующей поиск по лексическим областям вызова:

```
function say(myName) {
    var greeting = "Hello";
    output();

    function output() {
        console.log(
            `${ greeting }, ${ myName }!`
        );
    }
}

say("Kyle");
// Hello, Kyle!
```

Внутренняя функция `output()` обращается к переменным `greeting` и `myName` из окружающей области видимости. Но вызов `output()` происходит в той же области видимости, в которой, конечно, `greeting` и `myName` все еще доступны; это просто лексическая область видимости, а не замыкание.

Любой язык с лексической областью видимости, функции которого не поддерживают замыкания, будет вести себя так же.

Более того, переменные глобальной области видимости не могут (наблюдаемо) участвовать в замыканиях, потому что они всегда доступны в любой точке. Никакая функция не может вызываться в любой части цепочки областей видимости, которая бы не являлась потомком глобальной области видимости.

Пример:

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getFirstStudent() {
    return function firstStudent(){
        return students[0].name;
    };
}

var student = getFirstStudent();

student();
// Kyle
```

Внутренняя функция `firstStudent()` обращается к `students` — переменной за пределами ее собственной области видимости. Но так как `students` находится в глобальной области видимости, неважно, где эта функция вызывается в программе; ее способность обращаться к `students` — не что иное, как обычная лексическая область видимости.

Все вызовы функций могут обращаться к глобальным переменным независимо от того, поддерживаются ли замыкания языком или нет. Глобальные переменные просто не нужно включать в замыкания.

Переменные, которые просто присутствуют, но к которым не происходят обращения, не приводят к созданию замыканий:

```
function lookupStudent(studentID) {
    return function nobody(){
        var msg = "Nobody's here yet.";
        console.log(msg);
    };
}

var student = lookupStudent(112);

student();
// Nobody's here yet.
```

Внутренняя функция `nobody()` не замыкается ни по каким внешним переменным — она использует только свою собственную переменную `msg`. И хотя `studentID` присутствует в окружающей области видимости, `nobody()` не обращается к `studentID`. Движку JS не нужно поддерживать существование `studentID` после завершения выполнения `lookupStudent(..)`, поэтому система сборки мусора захочет освободить эту память!

Эта программа вела бы себя одинаково независимо от того, поддерживают ли функции JS замыкания или нет. Следовательно, наблюдаемых замыканий здесь нет.

При отсутствии вызова функции замыкание также не наблюдается:

```
function greetStudent(studentName) {
    return function greeting(){
        console.log(
            `Hello, ${ studentName }!`
        );
    };
}

greetStudent("Kyle");

// ничего не происходит
```

С этим примером все хитрее, потому что внешняя функция определенно вызывается. Но именно внутренняя функция *может* иметь замыкание, однако она никогда не вызывается; возвращенная функция здесь просто теряется. Таким образом, хотя формаль-

но движок JS создает замыкание на очень короткое время, оно не наблюдается в программе никаким осмысленным образом.

Возможно, дерево упало... Но мы этого не слышали, поэтому нас это совершенно не интересует.

### Наблюдаемое определение

Теперь все готово для определения замыкания.

Замыкание наблюдается тогда, когда функция использует переменную(-ые) из другой(-их) области(-ей) видимости даже при выполнении в области видимости, в которой эта(-и) переменная(-ые) должна(ы) быть недоступна(-ы).

Ключевые части определения:

- в замыкании должна быть задействована функция;
- она должна обращаться хотя бы к одной переменной из внешней области видимости;
- функция должна вызываться из другой ветви цепочки областей видимости относительно той, в которой находи(-я)тся переменная(-ые).

Определение, ориентированное на наблюдаемость, означает, что мы не должны отбрасывать замыкания, как какую-то теоретическую, не имеющую практической ценности безделушку. Вместо этого следует выявлять и планировать прямые, конкретные эффекты замыканий для поведения наших программ.

## Жизненный цикл замыканий и сборка мусора (GC)

Так как замыкания неразрывно связаны с экземпляром функции, ее замыкание по переменной продолжается до тех пор, пока существует ссылка на эту функцию.

Если десять функций замыкаются по одной и той же переменной и со временем девять из этих ссылок на функции исчезают, то одна

оставшаяся ссылка на функцию продолжит сохранять эту переменную в памяти. Как только последняя ссылка на функцию будет потеряна, последнее замыкание по этой переменной пропадает и сама переменная освобождается сборщиком мусора.

Это имеет важные последствия для построения эффективных и производительных программ. Замыкание может неожиданно помешать освобождению переменной, с которой вы уже завершили работу, что приводит к утечке памяти со временем. Вот почему важно освобождать ссылки на функции (а следовательно, и их замыкания), когда они перестают быть ненужными.

Пример:

```
function manageBtnClickEvents(btn) {
    var clickHandlers = [];

    return function listener(cb){
        if (cb) {
            let clickHandler =
                function onClick(evt){
                    console.log("clicked!");
                    cb(evt);
                };
            clickHandlers.push(clickHandler);
            btn.addEventListener(
                "click",
                clickHandler
            );
        }
        else {
            // passing no callback unsubscribes
            // all click handlers
            for (let handler of clickHandlers) {
                btn.removeEventListener(
                    "click",
                    handler
                );
            }

            clickHandlers = [];
        }
    };
}
```

```
// var mySubmitBtn = ..
var onSubmit = manageBtnClickEvents(mySubmitBtn);

onSubmit(function checkout(evt){
    // обработка оформления заказа
});

onSubmit(function trackAction(evt){
    // регистрация действия для аналитики
});

// позднее отменить регистрацию всех обработчиков:
onSubmit();
```

В этой программе внутренняя функция `onClick(..)` поддерживает замыкание по полученному значению `cb` (предоставляемый обратный вызов события). Это означает, что ссылки на функциональные выражения `checkout()` и `trackAction()` удерживаются посредством замыкания (и не могут быть освобождены сборщиком мусора), пока эти обработчики событий остаются зарегистрированными для подписки.

Когда мы вызываем `onSubmit()` без входных данных в последней строке, подписка всех обработчиков событий отменяется и массив `clickHandlers` очищается. После того как все ссылки на функции-обработчики освобождаются, замыкания ссылок `cb` на `checkout()` и `trackAction()` освобождаются.

При анализе общего рабочего состояния и эффективности программы отмена подписки на обработчик событий, когда надобность в нем отпадет, может быть даже важнее исходной подписки!

### На уровне переменных или на уровне области видимости?

Другой вопрос, которому необходимо уделить внимание: следует ли считать, что замыкание применяется только к тем внешним переменным, к которым вы обращаетесь, или же оно сохраняет всю цепочку областей видимости со всеми ее переменными?

Иначе говоря, в предыдущем фрагменте с подпиской на события замыкается ли внутренняя функция `onClick(..)` только на `cb` или же она также замыкается на `clickHandler`, `clickHandlers` и `btn`?

На концептуальном уровне замыкание работает на уровне переменных, а не области видимости. Обычно считается, что обратные вызовы Ajax, обработчики событий и все остальные формы функциональных замыканий обычно замыкаются только по тем переменным, к которым они явно обращаются.

Но реальность немного сложнее.

Рассмотрим еще один пример:

```
function manageStudentGrades(studentRecords) {
    var grades = studentRecords.map(getGrade);

    return addGrade;

    // ************************

    function getGrade(record){
        return record.grade;
    }

    function sortAndTrimGradesList() {
        // sort by grades, descending
        grades.sort(function desc(g1,g2){
            return g2 - g1;
        });

        // only keep the top 10 grades
        grades = grades.slice(0,10);
    }

    function addGrade(newGrade) {
        grades.push(newGrade);
        sortAndTrimGradesList();
        return grades;
    }
}
```

```
var addNextGrade = manageStudentGrades([
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    // ..еще много записей..
    { id: 6, name: "Sarah", grade: 91 }
]);

// позднее
addNextGrade(81);
addNextGrade(68);
// [ .., .., ... ]
```

Внешняя функция `manageStudentGrades(..)` получает список записей с данными студентов и возвращает ссылку на функцию `addGrade(..)`, которой мы присвоим внешнюю метку `addNextGrade(..)`. При каждом вызове `addNextGrade(..)` с новой оценкой мы получаем текущий список 10 наивысших оценок, отсортированный по убыванию (см. `sortAndTrimGradesList()`).

От конца исходного вызова `manageStudentGrades(..)` и между несколькими вызовами `addNextGrade(..)` переменная `grades` сохраняется внутри `addGrade(..)` посредством замыканий; так поддерживается текущий список наивысших оценок. Помните: в замыкание включается сама переменная `grades`, а не содержащийся в ней массив.

Впрочем, это не единственное задействованное замыкание. Заметите ли вы другие переменные, включенные в замыкание?

Вы заметили, что `addGrade(..)` обращается к переменной `sortAndTrimGradesList`? Это означает, что она также замыкается по этому идентификатору, в котором хранится ссылка на функцию `sortAndTrimGradesList()`. Вторая внутренняя функция должна продолжить существование, чтобы функция `addGrade(..)` могла продолжать вызывать ее, а это означает, что все замыкаемые *ей* переменные тоже должны сохраняться, хотя в этом конкретном случае ничего лишнего в замыкание не включается.

Что еще включается в замыкание?

Возьмем переменную `getGrade` (и ее функцию); она включается в замыкание? Обращение к ней происходит во внешней области видимости `manageStudentGrades(..)` при вызове `.map(getGrade)`. При этом к ней нет обращений из `addGrade(..)` или `sortAndTrimGradesList()`.

Как насчет (потенциально) большого списка записей студентов, передаваемого в `studentRecords`? Эта переменная включается в замыкание? Если включается, то массив записей студентов никогда не будет освобожден сборщиком мусора и в результате программа будет расходовать больше памяти, чем можно было ожидать. Но если присмотреться повнимательнее, ни одна из внутренних функций не обращается к `studentRecords`.

В соответствии с определением замыканий на уровне переменных, так как внутренние функции не обращаются к `getGrade` и `studentRecords`, они не включаются в замыкание. Они должны быть доступны для сборки мусора непосредственно после завершения вызова `manageStudentGrades(..)`.

Попробуйте отладить этот код в новом движке JS (например, v8 в Chrome) и установить точку прерывания в функции `addGrade(..)`. Возможно, вы заметите, что в инспекторе отсутствует переменная `studentRecords`. Это служит доказательством (по крайней мере, с точки зрения отладки), что движок не хранит `studentRecords` через замыкание.

Но в какой мере это наблюдение может служить доказательством? Возьмем следующую (довольно неестественную) программу:

```
function storeStudentInfo(id,name,grade) {
    return function getInfo(whichValue){
        // предупреждение:
        // использовать `eval(..)` не рекомендуется!
        var val = eval(whichValue);
        return val;
    };
}

var info = storeStudentInfo(73,"Suzy",87);
```

```
info("name");
// Suzy
info("grade");
// 87
```

Обратите внимание: внутренняя функция `getInfo(..)` не включает явно в замыкание переменные `id`, `name` или `grade`. Но вызовы `info(..)` вроде бы все равно могут обращаться к переменным, хотя и через трюк с лексической областью видимости с `eval(..)` (см. главу 1).

Получается, что все переменные определенно сохраняются через замыкание, хотя и внутренняя функция не содержит явных обращений к ней. Выходит, это опровергает утверждение о замыканиях *уровня переменных* в пользу замыканий *уровня областей видимости*? Зависит от обстоятельств.

Многие современные движки JS применяют *оптимизацию*, которая исключает из области видимости замыкания любые переменные, к которым отсутствуют явные обращения. Но как показывает пример с `eval(..)`, в некоторых ситуациях такая оптимизация неприменима и область видимости замыкания включает все исходные переменные. Другими словами, замыкание должно существовать на *уровне области видимости* (в зависимости от реализации), после чего необязательная оптимизация усекает область видимости до замыкаемых переменных (результат, эквивалентный замыканиям с *уровнем переменных*).

Даже несколько лет назад многие движки JS не применяли эту оптимизацию; возможно, ваши веб-сайты все еще работают в таких браузерах (особенно на старых или малопроизводительных устройствах). Это означает, что замыкания с долгим сроком жизни (например, обработчики событий) могут находиться в памяти намного дольше, чем можно было бы ожидать.

И тот факт, что речь идет о необязательной оптимизации, а не о требовании спецификации, означает, что мы не должны строить излишних ожиданий относительно ее применимости.

В тех случаях, когда переменная из области видимости замыкания содержит большое значение (например, объект или массив) и вы не хотите, чтобы эта память удерживалась, будет безопаснее (с точки зрения расходования памяти) вручную освободить значение, вместо того чтобы полагаться на оптимизацию замыканий/сборку мусора.

Внесем изменение в более ранний пример `manageStudentGrades(..)`, чтобы потенциально большой массив из `studentRecords` не удерживался в области видимости замыкания без необходимости:

```
function manageStudentGrades(studentRecords) {
    var grades = studentRecords.map(getGrade);

    // сбросить значение `studentRecords`, чтобы предотвратить
    // нежелательное удержание памяти в замыкании

    studentRecords = null;

    return addGrade;
    // ..
}
```

Мы не удаляем `studentRecords` из области видимости замыкания — это не в наших силах. Мы гарантируем, что даже если `studentRecords` останется в области видимости замыкания, эта переменная уже не ссылается на потенциально большой массив данных; массив может быть уничтожен в ходе сборки мусора.

Еще раз: во многих случаях JS может автоматически оптимизировать программу для достижения того же эффекта. Тем не менее стоит обзавестись полезной привычкой и явно следить за тем, чтобы значительный объем памяти устройства не оставался зарезервированным дольше, чем необходимо.

Собственно говоря, формально функция `getGrade()` также не нужна нам после завершения вызова `.map(getGrade)`. Если профилирование приложения показывает, что затраты памяти критичны для приложения, мы можем освободить еще немного памяти, освобождая эту ссылку, чтобы ее значение не оставалось в памяти. Скорее всего, это совершенно излишне в этом учебном

примере, но стоит знать этот общий прием, если вы занимаетесь оптимизацией затрат памяти в приложениях.

Вывод: важно знать, где в наших программах появляются замыкания и какие переменные в них включаются. Следует тщательно управлять этими замыканиями, чтобы в памяти удерживался минимально необходимый объем памяти, а память не расходовалась понапрасну.

## Альтернативная точка зрения

В нашем рабочем определении замыканий предполагается, что эти функции являются «полноправными значениями», которые могут передаваться в программах, как и любые другие значения. Замыкание представляет собой ссылочную связь, которая соединяет эту функцию с областями видимости/переменными за ее пределами независимо от того, где находится эта функция.

Вспомним приводившийся ранее в этой главе пример кода с цветовой пометкой областей видимости:

```
// внешняя/глобальная область видимости: КРАСНЫЙ(1)

function adder(num1) {
    // function scope: СИНИЙ(2)

    return function addTo(num2){
        // function scope: ЗЕЛЕНЫЙ(3)

        return num1 + num2;
    };
}

var add10To = adder(10);
var add42To = adder(42);

add10To(15); // 25
add42To(9); // 51
```

Наша текущая точка зрения предполагает, что при каждой передаче и вызове функции замыкание сохраняет скрытую ссылку на

исходную область видимости, чтобы упростить доступ к переменным из замыкания. На рис. 4, повторенном для удобства, показан этот принцип.


**Рис. 4 (*повторение*).** Наглядное представление замыканий

Но есть другой взгляд на замыкания — точнее, на природу передаваемых функций, который может углубить ваши ментальные модели.

Эта альтернативная модель уводит на второй план концепцию функций как полноправных значений и вместо этого уделяет основное внимание тому, как функции (как и все непримитивные значения) хранятся в JS по ссылке и присваиваются/передаются копированием ссылки; за дополнительной информацией обращайтесь к приложению А книги «Познакомьтесь, JavaScript».

Чтобы не рассматривать перемещение экземпляра внутренней функции `addTo(..)` во внешнюю область видимости КРАСНЫЙ (1) командой `return` и присваиванием, мы можем представить, что экземпляры функций на самом деле остаются на месте со своей собственной средой области видимости — конечно, с сохранением цепочки областей видимости.

В область видимости КРАСНЫЙ (1) передается только ссылка на экземпляр функции, а не сам экземпляр функции. На рис. 5

изображены экземпляры внутренней функции, остающиеся на своих местах, на которые указывают ссылки КРАСНЫЙ (1) `addTo10` и `addTo42` соответственно:

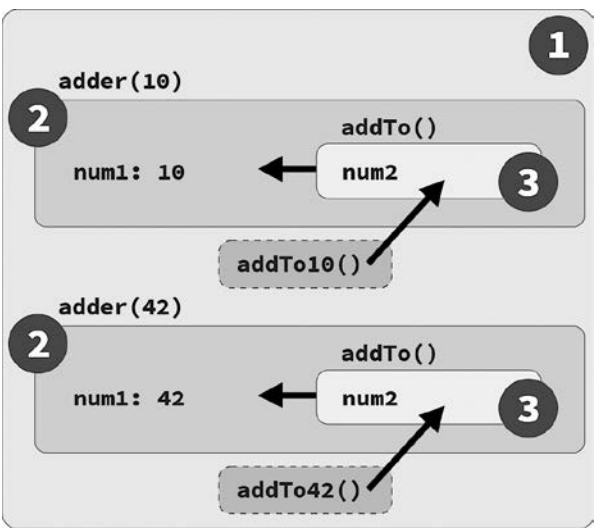


**Рис. 5.** Наглядное представление замыканий (альтернативное)

Как показано на рис. 5, каждый вызов `adder(..)` все еще создает новую область видимости СИНИЙ (2), содержащую переменную `num1`, а также экземпляр области видимости ЗЕЛЕНЫЙ (3) `addTo(..)`. Но в отличие от рис. 4, теперь эти экземпляры ЗЕЛЕНЫЙ (3) остаются на месте, оставаясь вложенными естественным образом в свои экземпляры области видимости СИНИЙ (2). Во внешнюю область видимости КРАСНЫЙ (1) перемещаются ссылки `addTo10` и `addTo42`, а не сами экземпляры функций.

При вызове `addTo10(15)` вызывается экземпляр функции `addTo(..)` (все еще остающийся на месте в своей исходной области видимости СИНИЙ (2)). Так как сам экземпляр функции никуда не перемещается, конечно, он сохраняет естественный доступ к своей цепочке областей видимости. То же относится к вызову `add-`

`To42(9)` — в нем нет ничего необычного, выходящего за рамки лексической видимости.

Тогда что же такое замыкание, если не *волшебство*, которое позволяет функции поддерживать ссылку на исходную цепочку областей видимости, даже если эта функция перемещается в другие области видимости? В этой альтернативной модели функции остаются на месте и продолжают обращаться к своей исходной цепочке областей видимости, которая всегда была для них доступна.

Здесь замыкание скорее становится *волшебством*, которое позволяет **поддерживать существование экземпляра функции** вместе со всей его областью видимости и цепочкой, пока в программе остается хотя бы одна ссылка на этот экземпляр функции, существующий в любой другой части программы.

Такое определение замыкания в меньшей степени ориентировано на наблюдаемые эффекты и звучит чуть менее знакомо по сравнению с традиционными академическими представлениями. Тем не менее оно остается полезным, потому объяснение сути замыкания упрощается до прямолинейной комбинации ссылок и экземпляров функций, остающихся на своих местах.

Нельзя сказать, что предыдущая модель (рис. 4) ошибочно описывает замыкание в JS. Просто она чуть более концептуальна — это теоретическая точка зрения на замыкания. С другой стороны, альтернативную модель (рис. 5) можно описать как в большей степени ориентированную на реализацию — на то, как реально работает JS.

Обе точки зрения/модели полезны для понимания замыканий, но, возможно, какая-то из них покажется читателю более понятной. Какую бы вы ни выбрали, наблюдаемые эффекты в программе остаются одними и теми же.

Альтернативная модель замыканий влияет на то, можно ли отнести синхронные обратные вызовы к примерам замыканий или нет. Подробнее этот нюанс рассматривается в приложении А.

## Для чего нужны замыкания?

Теперь, когда вы получили разностороннее представление о том, что такое замыкания и как они работают, рассмотрим некоторые возможности того, как они могут улучшить структуру кода и организацию программ-примеров.

Представьте, что на странице находится кнопка, которая при нажатии должна прочитать и отправить некоторые данные при помощи запроса Ajax.

Без использования замыканий:

```
var APIendpoints = {
    studentIDs:
        "https://some.api/register-students",
    // ..
};

var data = {
    studentIDs: [ 14, 73, 112, 6 ],
    // ..
};

function makeRequest(evt) {
    var btn = evt.target;
    var recordKind = btn.dataset.kind;
    ajax(
        APIendpoints[recordKind],
        data[recordKind]
    );
}

// <button data-kind="studentIDs">
// Register Students
// </button>
btn.addEventListener("click",makeRequest);
```

Функция `makeRequest(..)` получает от события щелчка только объект `evt`. Она должна получить атрибут `data-kind` целевого элемента кнопки и использовать это значение для получения как

URL-адреса для конечной точки API, так и данных, которые должны быть включены в запрос Ajax.

Такое решение работает, но, к сожалению, обработчик события должен читать атрибут DOM при каждом срабатывании. Почему бы обработчику события не запомнить это значение? Попробуем воспользоваться замыканием для улучшения кода:

```
var APIendpoints = {
    studentIDs:
        "https://some.api/register-students",
    // ..
};

var data = {
    studentIDs: [ 14, 73, 112, 6 ],
    // ..
};

function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;
    btn.addEventListener(
        "click",
        function makeRequest(evt){
            ajax(
                APIendpoints[recordKind],
                data[recordKind]
            );
        }
    );
}

// <button data-kind="studentIDs">
// Register Students
// </button>

setupButtonHandler(btn);
```

С подходом `setupButtonHandler(..)` атрибут `data-kind` читается только один раз, а затем присваивается переменной `recordKind` при начальной инициализации. Затем `recordKind` включается в замыкание для внутреннего обработчика `makeRequest(..)`, и его

значение используется при каждой выдаче события для поиска URL и данных, которые требуется отправить.

Объект `evt` все еще передается `makeRequest(..)`, хотя мы его более не используем. Он все еще указывается для сохранения логической целостности с предыдущим фрагментом.

Размещая `recordKind` внутри `setupButtonHandler(..)`, мы ограничиваем раскрытие этой переменной более подходящим подмножеством программы; ее глобальное хранение ухудшило бы структуру и удобочитаемость кода. Замыкание позволяет экземпляру внутренней функции `makeRequest()` запомнить эту переменную и обращаться к ней при необходимости.

Развивая этот паттерн, мы можем определить URL и данные одновременно при инициализации:

```
function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;
    var requestURL = APIendpoints[recordKind];
    var requestData = data[recordKind];

    btn.addEventListener(
        "click",
        function makeRequest(evt){
            ajax(requestURL,requestData);
        }
    );
}
```

Теперь `makeRequest(..)` замыкается на `requestURL` и `requestData`; такое решение немного проще понять, и оно также обладает чуть лучшим быстродействием.

Два сходных приема из парадигмы функционального программирования (FP), зависящие от замыканий, — частичное применение и каррирование. Вкратце: в этих приемах изменяется форма функций, получающих несколько входных значений, чтобы некоторые входные данные предоставлялись заранее, а другие — позднее; исходные входные данные запоминаются через замыкание. После

того как все входные данные будут предоставлены, выполняется соответствующее действие.

Создавая экземпляр функции, который инкапсулирует некоторую информацию (посредством замыкания), функция с хранимой информацией может использоваться позднее напрямую без повторной передачи этих входных данных. Это способствует упрощению кода, а также предоставляет возможность назначения частично применяемым функциям более содержательных семантических имен.

Адаптированная версия частичного применения позволяет дополнительно улучшить этот код:

```
function defineHandler(requestURL,requestData) {
    return function makeRequest(evt){
        ajax(requestURL,requestData);
    };
}

function setupButtonHandler(btn) {
    var recordKind = btn.dataset.kind;
    var handler = defineHandler(
        APIendpoints[recordKind],
        data[recordKind]
    );
    btn.addEventListener("click",handler);
}
```

Входные данные `requestURL` и `requestData` предоставляются заранее, в результате чего создается частично примененная функция `makeRequest(..)`, которой мы присваиваем локальную метку `handler`. Когда событие сработает, `handler()` передается последнее входное значение (`evt`, хотя в данном случае оно игнорируется), и после сбора всех входных данных инициируется запрос Ajax.

В отношении поведения эта программа очень похожа на предыдущую, и она использует такой же тип замыкания. Однако выделение создания `makeRequest()` в отдельную вспомогательную функцию (`defineHandler(..)`) упрощает повторное использование этого определения в программе. Мы также явно ограничиваем область видимости замыкания двумя необходимыми переменными.

## Напоследок о замыканиях

Приближаясь к концу довольно насыщенной главы, переведите дыхание и попытайтесь осознать все сказанное. Не каждому под силу усвоить такой объем информации за раз!

В этой главе были рассмотрены две концептуальные модели замыканий:

- Основанная на наблюдаемости: замыкание — экземпляр функции, запоминающей свои внешние переменные даже при ее передаче и вызове в других областях видимости.
- Основанная на реализации: замыкание — экземпляр функции и окружение ее области видимости, хранящиеся «на месте»; ссылки на эту функцию передаются и вызываются в других областях видимости.

Основные преимущества замыканий для наших программ.

- Замыкание может повысить эффективность, так как экземпляр функции может запомнить ранее определенную информацию, вместо того чтобы каждый раз вычислять ее заново.
- Замыкания могут улучшить удобочитаемость кода. Они ограничивают раскрытие за счет инкапсуляции переменной(-ых) внутри экземпляров функции, при этом сохраняя на будущее возможность доступа к информации в этих переменных. С полученными более узкими, более специализированными экземплярами функций проще взаимодействовать, так как хранимую информацию не нужно передавать при каждом вызове.

Прежде чем двигаться дальше, выделите немного времени на то, чтобы пересказать итоги *своими словами*. Объясните, что такое замыкание и почему оно может пригодиться в ваших программах. А нам осталась последняя глава, в которой на базе замыканий будет построен паттерн «Модуль».

# 8 Паттерн «Модуль»

Эта глава завершается рассмотрением одного из важнейших паттернов организации кода во всем программировании — *модулей*. Как вы увидите, модули по своей сути строятся на том материале, который приводился ранее: ваши усилия по изучению лексической видимости и замыканий окупятся лишний раз.

Мы рассмотрели все тонкости лексической видимости, от широты глобальной области видимости до глубин вложенных блоковых областей и тонкостей жизненного цикла переменных. Затем концепция лексической видимости была использована для представления всей мощи замыканий.

Выделите немного времени и поразмыслите над тем, как далеко вы зашли в своем путешествии; вы упорно шли вперед, чтобы лучше узнать JS!

Центральная тема этой книги заключается в том, что понимание и уверенное владение областями видимости и замыканиями играют ключевую роль в правильном структурировании и организации вашего кода, особенно в решениях относительно того, где следует хранить информацию в переменных.

В этой последней главе я покажу, как модули воплощают важность этих тем, поднимая их от абстрактных концепций до конкретных, практических усовершенствований при построении программ.

## Инкапсуляция и принцип наименьшего раскрытия (POLE)

Инкапсуляция часто представляется как принцип объектно-ориентированного (ОО) программирования, но это намного более фундаментальная концепция, находящая широкое практическое применение. Цель инкапсуляции — упаковка, т. е. совместное размещение, информации (данных) и поведения (функций), служащих общей цели.

Независимо от синтаксиса или программных механизмов в простейшем виде суть инкапсуляции можно представить как использование отдельных файлов для хранения частей общей программы, объединенных единым предназначением. Если упаковать все, что относится к списку результатов поиска, в один файл с именем search-list.js, вы тем самым инкапсулируете эту часть программы.

Последние тенденции в современном фронтенд-программировании к организации приложений на базе компонентной архитектуры способствуют еще более широкому применению инкапсуляции. Для многих разработчиков кажется естественным объединить все, что относится к списку результатов поиска (не только код, но и разметку представления и стилевое оформление), в единицу программной логики — иногда вполне материальную, с которой можно взаимодействовать. И затем мы называем этот набор компонентом `SearchList`.

Другая ключевая цель — управление видимостью некоторых аспектов инкапсулированных данных и функциональности. Вспомните принцип наименьшего раскрытия (POLE) из главы 6, направленный на защиту от различных *рисков* чрезмерного раскрытия областей видимости; эти риски распространяются как на переменные, так и на функции. В JS управление видимостью обычно реализуется через механизм лексической области видимости.

Идея заключается в том, чтобы сгруппировать взаимосвязанные части программы и избирательно ограничить программный доступ к тем частям, которые мы считаем *приватными* подробностями

реализации. Те части, которые не считаются *приватными*, помечаются как *открытые* и доступные для всей программы.

Естественным следствием такой работы становится улучшение организации кода. Программный продукт проще строить и сопровождать, если вы знаете, где что находится, с четкими и очевидными границами и точками соединения. Также вам будет проще обеспечивать качество кода, если вы будете избегать опасностей чрезмерного раскрытия данных и функциональности.

## Что такое модуль?

Модуль представляет собой набор взаимосвязанных данных и функций (часто называемых методами в этом контексте), характеризуемых четким разделением между *приватными* подробностями реализации и *открытыми* аспектами, обычно называемыми открытым API.

Модуль также *обладает состоянием*: он поддерживает некоторую информацию во времени вместе с функциональностью для чтения и обновления этой информации.

В более широком смысле паттерн «Модуль» обеспечивает модуляризацию на системном уровне за счет слабых связей и других средств программной архитектуры. Это сложная тема, выходящая за рамки нашего обсуждения, но она заслуживает того, чтобы вы самостоятельно изучили ее.

Чтобы вы лучше поняли, что такое модуль, сравним некоторые характеристики модулей с полезными паттернами программирования, которые модулями не являются.

### Пространства имен (группировка без состояния)

Если вы группируете несколько взаимосвязанных функций без данных, то такая группировка не обеспечивает той инкапсуляции,

которую подразумевают модули. Для подобной группировки функций *без состояния* существует специальный термин — «пространство имен»:

```
// пространство имен, не модуль
var Utils = {
    cancelEvt(evt) {
        evt.preventDefault();
        evt.stopPropagation();
        evt.stopImmediatePropagation();
    },
    wait(ms) {
        return new Promise(function c(res){
            setTimeout(res,ms);
        });
    },
    isValidEmail(email) {
        return /[^@]+@[^@.]+\.[^@.]+/.test(email);
    }
};
```

`Utils` — полезный набор вспомогательных функций, но все эти функции не зависят от состояния. Группировка функциональности обычно является полезной практикой, но от этого она не становится модулем. Вместо этого мы определили пространство имен `Utils` и упорядочили в нем функции.

### Структуры данных (группировка с состоянием)

Даже если вы группируете данные вместе с функциями, обладающими состоянием, но не ограничиваете их видимости, вы не достигаете аспекта инкапсуляции, определяемого принципом наименьшего раскрытия (POLE); вряд ли полученная конструкция заслуживает название «модуль».

Пример:

```
// структура данных, не модуль
var Student = {
    records: [
        { id: 14, name: "Kyle", grade: 86 },
```

```
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ],
    getName(studentID) {
        var student = this.records.find(
            student => student.id == studentID
        );
        return student.name;
    }
};

Student.getName(73);
// Suzy
```

Так как `records` содержит общедоступные данные, не скрытые ни за каким открытым API, `Student` в действительности не является модулем.

`Student` обладает аспектом данные + функциональность, присущим инкапсуляции, но не обладает аспектом управления видимостью. Для таких случаев лучше подходит термин «структура данных».

## Модули (управление доступом с состоянием)

Чтобы дух паттерна «Модуль» воплотился в полной мере, понадобятся не только группировка и состояние, но и управление доступом через видимость (приватные/открытые части).

Преобразуем структуру `Student` из предыдущего раздела в модуль. Начнем с формы, которую я называю классическим модулем (когда она только появилась в начале 2000-х, изначально использовался термин «модуль с управлением видимостью»). Пример:

```
var Student = (function defineStudent(){
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ];
```

```
    var publicAPI = {
        getName
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find(
            student => student.id == studentID
        );
        return student.name;
    }
})();

Student.getName(73); // Suzy
```

Здесь структура `Student` является примером модуля. Она предоставляет открытый API с единственным методом `getName(..)`. Этот метод обращается к приватным скрытым данным `records`.

Должен указать, что данные студентов, явно запрограммированные в этом определении модуля, приведены только для наглядности. Типичный модуль в вашей программе получает данные из внешнего источника — обычно из баз данных, файлов данных JSON, вызовов Ajax и т. д. Затем эти данные внедряются в экземпляр модуля (как правило, методами открытого API модуля).

Как работает классический формат модулей?

Обратите внимание: экземпляр модуля создается выполняемым IIFE `defineStudent()`. IIFE возвращает объект (с именем `publicAPI`), который содержит свойство со ссылкой на внутреннюю функцию `getName(..)`.

Присваивание объекту имени `publicAPI` определяется моими стилистическими предпочтениями. Объекту можно присвоить любое имя на ваш выбор (для JS это никакой роли не играет) или вы можете вернуть объект напрямую без присваивания внутрен-

ней именованной переменной. Подробнее этот выбор рассматривается в приложении А.

С внешней точки зрения `Student.getName(..)` вызывает эту предоставленную внутреннюю функцию, которая сохраняет доступ к внутренней переменной `records` через замыкание.

Вы не обязаны возвращать объект, одним из свойств которого является функция. Также функцию можно вернуть напрямую, вместо объекта — все основные составляющие классического модуля при этом сохраняются.

Из-за особенностей работы лексических областей видимости определения переменных и функций внутри функции определения внешнего модуля по умолчанию становятся приватными. Только свойства, добавленные в открытый объект API, возвращаемый функцией, будут экспортированы для внешнего открытого использования.

Использование IIFE подразумевает, что программе понадобится только один центральный экземпляр модуля — такой экземпляр обычно называется одиночкой (singleton). Рассмотренный пример достаточно прост, и нет никаких очевидных причин, из-за которых программе могли бы потребоваться сразу несколько экземпляров модуля `Student`.

### Фабрика модулей (множественные экземпляры)

Но что, если вы хотите определить модуль, который может существовать в программе в нескольких экземплярах? В таком случае придется слегка изменить код:

```
// фабричная функция, не IIFE для создания одиночного экземпляра
function defineStudent() {
    var records = [
        { id: 14, name: "Kyle", grade: 86 },
        { id: 73, name: "Suzy", grade: 87 },
        { id: 112, name: "Frank", grade: 75 },
        { id: 6, name: "Sarah", grade: 91 }
    ];
```

```
    var publicAPI = {
        getName
    };

    return publicAPI;

    // ************************

    function getName(studentID) {
        var student = records.find(
            student => student.id == studentID
        );
        return student.name;
    }
}

var fullTime = defineStudent();
fullTime.getName(73); // Suzy
```

Чтобы не определять `defineStudent()` в форме IIFE, мы просто определяем обычную автономную функцию, которая в данном контексте обычно называется фабрикой модулей.

Затем программа вызывает фабрику модулей для создания экземпляра модуля, которому присваивается имя `fullTime`. Этот экземпляр модуля подразумевает создание нового экземпляра внутренней области видимости, а следовательно, нового замыкания, в котором `getName(..)` удерживает `records`. Теперь `fullTime.getName(..)` вызывает метод для этого конкретного экземпляра.

## Определение классического модуля

Итак, какими же признаками должен обладать классический модуль?

- Должна существовать внешняя область видимости — обычно от функции-фабрики модулей, выполняемой хотя бы один раз.

- Внутренняя область видимости модуля должна содержать хотя бы один блок скрытой информации, представляющей состояние модуля.
- Модуль должен возвращать через свой открытый API ссылку на хотя бы одну функцию, которая содержит замыкание на скрытое состояние модуля (чтобы это состояние сохранялось после вызова).

Вероятно, вы столкнетесь с другими вариациями на тему классических модулей, которые будут более подробно рассмотрены в приложении А.

## Модули Node CommonJS

В главе 4 был представлен формат модулей CommonJS, используемых Node. В отличие от формата классических модулей, описанного выше, в котором вы можете упаковать фабрику модулей или IIFE наряду с любым другим кодом, включая другие модули, модули CommonJS соответствуют файлам — один модуль на файл.

Изменим наш пример модуля, чтобы он соответствовал этому формату:

```
module.exports.getName = getName;

// ************************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 }
];

function getName(studentID) {
    var student = records.find(
        student => student.id == studentID
    );
    return student.name;
}
```

Идентификаторы `records` и `getName` принадлежат области видимости верхнего уровня этого модуля, но не глобальной области видимости (см. главу 4). Как следствие, все составляющие *по умолчанию* являются приватными для этого модуля.

Чтобы открыть доступ к чему-либо через открытый API в модуле CommonJS, мы добавляем свойство в пустой объект, предоставляемый в виде `module.exports`. В старом унаследованном коде вам могут встретиться ссылки, в которых используется только `exports`, но для ясности кода всегда следует уточнять эту ссылку префиксом `module.`

По стилистическим соображениям я предпочитаю размещать `exports` в начале, а реализацию модуля — в конце файла. При этом команды с `exports` можно размещать где угодно. Настоятельно рекомендую группировать их в начале или конце файла.

Некоторые разработчики привыкли заменять объект `exports` по умолчанию:

```
// определение нового объекта для API
module.exports = {
    // ..exports..
};
```

У такого подхода есть некоторые нежелательные странности, включая непредсказуемое поведение при циклических зависимостях между модулями. По этой причине не рекомендую использовать замену объекта. Если вы хотите назначить сразу несколько элементов экспортирования одновременно, используя определение в стиле объектных литералов, это можно сделать так:

```
Object.assign(module.exports,{
    // .. exports ..
});
```

Что здесь происходит? При определении объектного литерала `{..}` с указанием открытого API вашего модуля с последующим `Object.assign(..)` происходит поверхностное копирование всех этих свойств в существующий объект `module.exports` вместо его

замены. Такое решение хорошо сочетает удобство с более безопасным поведением модуля.

Чтобы включить в модуль/программу другой экземпляр модуля, используйте метод Node `require(..)`. Если предположить, что этот модуль находится в файле /path/to/student.js, к нему можно обратиться следующим образом:

```
var Student = require("/path/to/student.js");

Student.getName(73);
// Suzy
```

После этого `Student` ссылается на открытый API модуля из нашего примера.

Модули CommonJS ведут себя как одиночные экземпляры по аналогии со стилем определения модулей IIFE, описанным ранее. Сколько бы раз вы ни вызывали `require(..)` для одного модуля, вы будете просто получать дополнительные ссылки на один общий экземпляр модуля.

Механизм `require(..)` работает по принципу «все или ничего»; он включает ссылку на весь предоставляемый открытый API модуля. Если вам нужно обратиться только к части API, типичное решение выглядит так:

```
var getName = require("/path/to/student.js").getName;
// альтернативный вариант:
var { getName } = require("/path/to/student.js");
```

По аналогии с форматом классических модулей явно экспортируемые методы API модуля CommonJS удерживают замыкания на внутренние подробности модулей. Так состояние модуля (одиночный экземпляр) поддерживается на протяжении жизненного цикла программы.

В командах Node `require("student")` неабсолютные пути (`"student"`) предполагают расширение `.js` и проводят поиск в `node_modules`.

## Современные модули ES (ESM)

Формат ESM в чем-то напоминает формат CommonJS. Он также базируется на файлах, экземпляры модулей являются одиночными, и по умолчанию все части считаются приватными. Одно заметное различие заключается в том, что файлы ESM всегда работают в строгом режиме и не требуют включения директивы `"use strict"` в начало файла. Модули ESM невозможно определить так, чтобы они выполнялись в нестрогом режиме.

Вместо `module.exports` в CommonJS ESM использует ключевое слово `export` для предоставления доступа к элементам через открытый API модуля. Ключевое слово `import` заменяет команду `require(..)`. Отредактируем students.js для использования формата ESM:

```
export getName;

// ***********************

var records = [
    { id: 14, name: "Kyle", grade: 86 },
    { id: 73, name: "Suzy", grade: 87 },
    { id: 112, name: "Frank", grade: 75 },
    { id: 6, name: "Sarah", grade: 91 }
];

function getName(studentID) {
    var student = records.find(
        student => student.id == studentID
    );
    return student.name;
}
```

Единственное изменение — команда `export getName`. Как и в предыдущем случае, команды `export` могут находиться в любой позиции файла, хотя команда `export` должна находиться в области видимости верхнего уровня; она не может располагаться внутри любого другого блока или функции.

ESM поддерживает разные формы записи команд `export` в ESM. Пример:

```
export function getName(studentID) {
    // ..
}
```

Хотя ключевому слову `function` предшествует `export`, эта форма остается объявлением функции, которая также дополнительно экспортируется. А именно идентификатор `getName` поднимается как функция (см. главу 5), поэтому он доступен во всей области видимости модуля.

Другой возможный вариант:

```
export default function getName(studentID) {
    // ..
}
```

Это так называемое экспортирование по умолчанию, которое отличается семантикой от других видов экспортирования. В сущности, экспортирование по умолчанию представляет собой сокращенную форму для потребителей модуля. Это более компактный синтаксис для тех, кому нужен только один элемент API по умолчанию.

Экспортируемые элементы, которые не экспортируются по умолчанию, называются именованными.

Ключевое слово `import` — как и `export`, оно должно использоваться только на верхнем уровне ESM за пределами любых блоков или функций — также существует в нескольких синтаксических разновидностях. Первый вариант называется именованным импортированием:

```
import { getName } from "/path/to/students.js";

getName(73); // Suzy
```

Как видите, эта форма импортирует из модуля только открытые элементы API с указанными именами (все, что не указано явно, пропускается). Эти идентификаторы добавляются в область видимости верхнего уровня текущего модуля. Такая разновидность

импортирования знакома всем, кто пользовался пакетным импортированием в таких языках, как Java.

В фигурных скобках `{..}` можно перечислить несколько элементов API, разделяя их запятыми. Именованный импортируемый элемент также можно переименовать при помощи ключевого слова `as`:

```
import { getName as getStudentName }
    from "/path/to/students.js";

getStudentName(73); // Suzy
```

Если функция `getName` является экспортированием по умолчанию для модуля, его можно импортировать следующим образом:

```
import getName from "/path/to/students.js";

getName(73); // Suzy
```

Единственное отличие — отсутствие `{ }` вокруг импортируемого элемента. Если вы хотите смешать импортирование по умолчанию с другими именованными импортируемыми элементами, это делается так:

```
import { default as getName, /* .. и т. д. .. */ }
    from "/path/to/students.js";

getName(73); // Suzy
```

Следующая форма импортирования называется импортированием пространства имен:

```
import * as Student from "/path/to/students.js";

Student.getName(73); // Suzy
```

Наверное, это очевидно, но символ * импортирует все экспортируемые элементы API (как по умолчанию, так и именованные) и сохраняет их под одним заданным идентификатором пространства имен. Такой подход ближе всего соответствует форме клас-

сических модулей, которые поддерживались на протяжении большей части истории JS.

На момент написания книги современные браузеры поддерживали ESM уже несколько лет, но более или менее стабильная поддержка ESM в Node появилась относительно недавно и с тех пор прошла довольно значительный путь. Вероятно, эволюция будет продолжаться в ближайшие годы; включение поддержки ESM в ES6 создало ряд нетривиальных проблем совместимости с системой взаимодействия с модулями CommonJS в Node. За всеми последними подробностями обращайтесь к документации ESM в Node:

https://nodejs.org/api/esm.html

## На выходе из области видимости

Независимо от того, какой формат вы используете — классический формат модулей (браузер или Node), формат CommonJS (в Node) или ESM (браузер или Node), модули остаются одним из самых эффективных способов структурирования и организации функциональности и данных вашей программы.

Паттерн «Модуль» завершает наше путешествие по этой книге. Вы учились применять правила лексических областей видимости для размещения переменных и функций в правильных позициях. Принцип наименьшего раскрытия диктует защитный подход *приватности по умолчанию*, с которым вы избегаете чрезмерного раскрытия и взаимодействуете только с минимально необходимой поверхностью открытого API.

А если заглянуть на более низкий уровень, *магия* с хранением состояния модулей основывается на замыканиях, использующих систему лексических областей видимости.

На этом основная часть книги завершается. Поздравляю, вы прошли долгий путь! Как я говорил уже не раз, сейчас самое время сделать паузу, поразмыслить и потренироваться в применении того, что вы узнали.

# А Дальнейшее изучение

В этом приложении рассматриваются некоторые нюансы и граничные случаи, связанные со многими темами, рассмотренными в основном тексте книги. Это необязательный вспомогательный материал.

Некоторые люди считают, что слишком глубокое погружение в граничные случаи и разнообразные мнения только создают лишний шум и отвлекают — по их мнению, разработчику лучше придерживаться исхоженных путей. Мой подход критиковали как непрактичный и не способствующий эффективной работе. Я понимаю эту точку зрения, хотя и не разделяю ее.

Я считаю, что лучше вооружиться знаниями о том, как работают те или иные механизмы, чем пропускать все подробности, руководствуясь догадками и недостатком любознательности. Рано или поздно вы столкнетесь с ситуациями, в которых что-то возникает из той части, которую вы еще не изучали. Дорога не всегда бывает ровной. Так почему бы не подготовиться к неизбежным ухабам и рытвинам на пересеченной местности?

Здесь мое мнение выражено сильнее, чем в основной части книги; помните об этом, когда будете обдумывать и запоминать эту информацию. Приложение немного напоминает подборку постов в микроблоге, посвященных отдельным темам книги. Обсуждение

будет длинным и непростым, так что не жалейте времени и не пролистывайте сложные моменты.

## Предполагаемые области видимости

Области видимости иногда создаются в неочевидных местах. На практике эти предполагаемые области видимости редко влияют на поведение вашей программы, и все же вам будет полезно знать об их существовании. Обращайте внимание на следующие неочевидные области видимости:

- область видимости параметров;
- область видимости имен функций.

### Область видимости параметров

Из метафоры диалога в главе 2 может создаться впечатление, что параметры функций по сути не отличаются от локально объявленных переменных в области видимости функции. Но это не всегда так.

Пример:

```
// внешняя/глобальная область видимости: КРАСНЫЙ (1)

function getStudentName(studentID) {
    // область видимости функции: СИНИЙ (2)

    // ..
}
```

Здесь `studentID` считается простым параметром, поэтому он ведет себя как элемент области видимости функции СИНИЙ (2). Но если преобразовать его в сложный параметр, технически это перестанет быть так. К сложным формам параметров относятся параметры со значениями по умолчанию, переменные списки параметров (с использованием `...`) и деструктурированные параметры.

Пример:

```
// внешняя/глобальная область видимости: КРАСНЫЙ (1)

function getStudentName(/*СИНИЙ (2)*/ studentID = 0) {
    // область видимости функции: ЗЕЛЕНЫЙ (3)

    // ..
}
```

Здесь список параметров фактически превращается в отдельную область видимости, а область видимости функции становится вложенной в эту область видимости.

Почему? Что от этого изменится? Сложные формы параметров создают различные граничные случаи, поэтому список параметров превращается в отдельную область видимости для повышения эффективности работы с ними.

Пример:

```
function getStudentName(studentID = maxID, maxID) {
    // ..
}
```

Будем считать, что операции выполняются слева направо. Значение по умолчанию `=maxID` для параметра `studentID` требует, чтобы переменная `maxID` уже существовала (и была инициализирована). Код выдает ошибку TDZ (глава 5). Причина в том, что переменная `maxID` объявлена в области видимости параметров, но еще не была инициализирована из-за порядка параметров. Если переключиться на противоположный порядок параметров, ошибка TDZ не возникнет:

```
function getStudentName(maxID,studentID = maxID) {
    // ..
}
```

Ситуация становится еще сложнее, если ввести функциональное выражение в позиции параметра по умолчанию, где оно может

создать собственное замыкание (глава 7) на параметрах предполагаемой области видимости:

```
function whatsTheDealHere(id,defaultID = () => id) {
    id = 5;
    console.log( defaultID() );
}

whatsTheDealHere(3);
// 5
```

Возможно, этот фрагмент имеет смысл, потому что стрелочная функция `defaultID()` замыкается на параметре/переменной `id`, которому затем присваивается значение 5. Но давайте добавим замещающее определение `id` в области видимости функции:

```
function whatsTheDealHere(id,defaultID = () => id) {
    var id = 5;
    console.log( defaultID() );
}

whatsTheDealHere(3);
// 3
```

Надо же! Объявление `var id = 5` замещает параметр `id`, но замыкание функции `defaultID()` распространяется на параметр, а не на замещающую переменную в теле функции. Это доказывает, что список параметров заключается в отдельную область видимости. Впрочем, и это еще не все!

```
function whatsTheDealHere(id,defaultID = () => id) {
    var id;

    console.log(`local variable 'id': ${ id }`);
    console.log(
        `parameter 'id' (closure): ${ defaultID() }`
    );

    console.log("reassigning 'id' to 5");
    id = 5;

    console.log(`local variable 'id': ${ id }`);
    console.log(
```

```
        `parameter 'id' (closure): ${ defaultID() }`
    );
}

whatsTheDealHere(3);
// local variable 'id': 3 <--- Что!? Невероятно!
// parameter 'id' (closure): 3
// reassigning 'id' to 5
// local variable 'id': 5
// parameter 'id' (closure): 3
```

Странно выглядит первое консольное сообщение. В этот момент замещающая локальная переменная `id` только что была объявлена конструкцией `var id`, которая, как было сказано в главе 5, обычно автоматически инициализируется `undefined` в начале области видимости. Почему не выводится `undefined`?

В этом конкретном граничном случае (ради совместимости с унаследованным кодом) JS автоматически инициализирует `id` не значением `undefined`, а значением параметра `id` (3)!

Хотя в данный момент кажется, что два идентификатора `id` представляют одну переменную, на самом деле это разные переменные (находящиеся в разных областях видимости).

Присваивание `id = 5` наглядно доказывает это расхождение; параметр `id` остается равным 3, а локальная переменная становится равной 5.

Чтобы не столкнуться с этими странностями, я рекомендую:

- никогда не замещать параметры локальными переменными;
- избегать использования функции с параметрами по умолчанию, которая создает замыкание по своим параметрам.

По крайней мере, теперь вы знаете о том, что список параметров находится в отдельной области видимости, если какие-либо из параметров не являются простыми, и сможете проявить необходимую осторожность.

# Область видимости имени функции

В разделе «Область видимости имени функции» главы 3 я упоминал о том, что имя функционального выражения добавляется в собственную область видимости функции. Вспомните:

```
var askQuestion = function ofTheTeacher(){
    // ..
};
```

Правда, `ofTheTeacher` не добавляется в окружающую область видимости (где объявляется `askQuestion`), но она также не *просто* добавляется в область видимости функции, как можно было бы ожидать. Это еще один странный граничный случай предполагаемой области видимости.

Идентификатор `name` функционального выражения принадлежит собственной предполагаемой области видимости, вложенной между внешней окружающей областью видимости и основной областью видимости внутренней функции.

Если бы переменная `ofTheTeacher` находилась в области видимости функции, можно было бы ожидать, что здесь произойдет ошибка:

```
var askQuestion = function ofTheTeacher(){
    // почему не происходит ошибка повторяющегося объявления?
    let ofTheTeacher = "Confused, yet?";
};
```

Форма объявления `let` не допускает повторного объявления (см. главу 5). Однако это абсолютно законное затенение, а не повторное объявление, потому что два идентификатора `ofTheTeacher` находятся в разных областях видимости.

Вряд ли вы когда-нибудь окажетесь в ситуации, в которой важна область видимости идентификатора имени функции. Но будет полезно знать, как работают эти механизмы. Чтобы не попасть в ловушку, никогда не замещайте идентификаторы имен функций.

## Анонимные и именованные функции

Как обсуждалось в главе 3, функции могут записываться как в именованной, так и в анонимной форме. Анонимная форма встречается гораздо чаще, но так ли это хорошо?

При выборе имен для ваших функций необходимо учитывать следующие факторы:

- автоматическое определение имен неполно;
- лексические имена допускают ссылки на самих себя;
- имена должны быть полезными описаниями;
- стрелочные функции не имеют лексических имен;
- IIFE также нужны имена.

### Явные или автоматически определяемые имена?

У каждой функции в вашей программе есть определенная цель. Если у функции нет цели, удалите ее, потому что она только попусту занимает место. Если цель *есть*, то для нее *можно* подобрать имя.

Вероятно, до этого момента большинство читателей соглашается со мной. Но означает ли это, что имя всегда следует включать в код? И вот здесь я предвижу немало удивленных взглядов. Я без малейших колебаний отвечу: да!

Прежде всего, anonymous в трассировке стека совершенно не упрощает процесс отладки:

```
btn.addEventListener("click",function(){
    setTimeout(function(){
        ["a",42].map(function(v){
            console.log(v.toUpperCase());
        });
    },100);
});
// Uncaught TypeError: не удается прочитать свойство
// 'toUpperCase' со значением null
```

```
// at myProgram.js:4
// at Array.map (<anonymous>)
// at myProgram.js:3
```

Сравните с результатом, который будет получен при назначении функциям имен:

```
btn.addEventListener("click",function onClick(){
    setTimeout(function waitAMoment(){
        ["a",42].map(function allUpper(v){
            console.log(v.toUpperCase());
        });
    },100);
});
// Uncaught TypeError: v.toUpperCase не является функцией
// at allUpper (myProgram.js:4)
// at Array.map (<anonymous>)
// at waitAMoment (myProgram.js:3)
```

Имена `waitAMoment` и `allUpper` присутствуют в выходных данных, благодаря чему трассировка стека предоставляет более содержательную информацию/контекст для отладки. Если вы присвоите разумные имена всем своим функциям, это упростит отладку программы.

Неприятная строка `<anonymous>`, которая все еще присутствует в результатах, относится к тому факту, что реализация `Array.map(..)` отсутствует в нашей программе, т к. встроена в движок JS. Дело не в какой-то путанице, возникшей в программе из-за сокращений для удобочитаемости.

Кстати, давайте убедимся в том, что мы одинаково понимаем смысл термина «именованная функция»:

```
function thisIsNamed() {
    // ..
}

ajax("some.url",function thisIsAlsoNamed(){
    // ..
});
```

```
var notNamed = function(){
    // ..
};

makeRequest({
    data: 42,
    cb /* тоже не имя */: function(){
        // ..
    }
});

var stillNotNamed = function butThisIs(){
    // ..
};
```

«Постойте-ка! — скажете вы. — Некоторые из этих функций *являются* именованными, верно?»

```
var notNamed = function(){
    // ..
};

var config = {
    cb: function(){
        // ..
    }
};

notNamed.name;
// notNamed

config.cb.name;
// cb
```

Такие имена называются *автоматически определяемыми*. В автоматически определяемых именах нет ничего плохого, но они не решают обсуждаемой проблемы.

### Отсутствующие имена?

Да, автоматически определяемые имена могут присутствовать в трассировке, что определенно лучше anonymous. Но...

```
function ajax(url,cb) {
    console.log(cb.name);
}

ajax("some.url",function(){
    // ..
});
// ""
```

Какая неприятность... Анонимные функциональные выражения, передаваемые как обратные вызовы, не могут получить автоматически определяемое имя, поэтому `cb.name` содержит только пустую строку `""`. Подавляющее большинство всех функциональных выражений, особенно анонимных, используется в аргументах обратных вызовов; имена им не присваиваются. Таким образом, возможности автоматического определения имен в лучшем случае ограниченны.

И автоматическое определение дает сбой не только с обратными вызовами:

```
var config = {};

config.cb = function(){
    // ..
};

config.cb.name;
// ""

var [ noName ] = [ function(){} ];
noName.name
// ""
```

Автоматическое определение имен также не справляется с любым присваиванием функционального выражения, которое не является *простым присваиванием*. Иначе говоря, если вы не будете

действовать осторожно и сознательно, почти все анонимные функциональные выражения в вашей программе не будут иметь никаких имен.

Автоматического определения имен попросту недостаточно.

И даже если функциональное выражение получит автоматически определяемое имя, оно все еще не считается полноценной именованной функцией.

### Кто я?

Без лексического идентификатора/имени функция не имеет внутренней возможности сослаться на саму себя. Автоссылки важны для таких механизмов, как рекурсия и обработка событий:

```
// не работает
runOperation(function(num){
    if (num <= 1) return 1;
    return num * oopsNoNameToCall(num - 1);
});

// тоже не работает
btn.addEventListener("click",function(){
    console.log("should only respond to one click!");
    btn.removeEventListener("click",oopsNoNameHere);
});
```

Исключение лексического имени из обратного вызова усложняет надежные автоссылки на функцию. Вы можете объявить в окружающей области видимости переменную, которая содержит ссылку на функцию, но эта переменная находится под контролем окружающей области видимости — ей можно присвоить новое значение и т. д., и поэтому она не настолько надежна, как функция, имеющая собственную внутреннюю автоссылку.

### Имена как дескрипторы

И последний (на мой взгляд, самый важный) момент: если имя функции не указано, читателю будет сложнее с первого взгляда

определить, для чего нужна эта функция. Чтобы разобраться в происходящем, ему придется читать больше кода, включая код внутри функции и окружающий код за пределами функции.

Пример:

```
[ 1, 2, 3, 4, 5 ].filter(function(v){
    return v % 2 == 1;
});
// [ 1, 3, 5 ]

[ 1, 2, 3, 4, 5 ].filter(function keepOnlyOdds(v){
    return v % 2 == 1;
});
// [ 1, 3, 5 ]
```

Вряд ли можно привести разумный аргумент в пользу того, что исключение имени `keepOnlyOdds` из первого обратного вызова более эффективно передает читателю цель этого обратного вызова. Вы экономите 13 символов, но теряете важную информацию, упрощающую чтение программы. Имя `keepOnlyOdds` предельно четко сообщает читателю, что происходит в программе.

Движок JS не обращает никакого внимания на имя. Зато оно важно для людей, читающих ваш код.

Сможет ли читатель взглянуть на команду `v % 2 == 1` и определить, что она делает? Безусловно. Но ему придется определять цель (и имя), выполняя код в уме. Даже краткая пауза замедляет чтение кода. С хорошим содержательным именем этот процесс проходит мгновенно и почти без усилий.

Взгляните на это так: сколько раз автору кода приходится определять цель функции перед добавлением имени в код? Наверно, один. Возможно, два или три, если имя потребуется изменить. Но сколько раз читателю кода придется вычислять это имя/цель? Каждый раз, когда он читает эту строку. Сотни раз? Тысячи? Больше?

Независимо от длины или сложности функции я считаю, что автор должен подобрать хорошее содержательное имя и включить его

в код. Имена должны присваиваться даже однострочным функциям в командах `map(..)` и `then(..)`:

```
lookupTheRecords(someData)
.then(function extractSalesRecords(resp){
    return resp.allSales;
})
.then(storeRecords);
```

Имя `extractSalesRecords` передает читателю смысл обработчика `then(..)` *лучше* любых попыток осознать его мысленным выполнением `return resp.allSales`.

Отсутствие имени функции может объясняться либо ленью (вы не хотите вводить несколько лишних символов), либо отсутствием творческого мышления (вы не можете придумать хорошее имя). Если вы не можете придумать хорошее имя, скорее всего, вы еще не понимаете саму функцию и ее цель. Возможно, функция плохо спроектирована или пытается решать слишком много задач и нуждается в переработке. Когда у вас появится хорошо спроектированная функция, специализированная для решения одной задачи, ее имя должно стать очевидным.

Обычно я использую такой прием: если во время первого написания функции я не полностью понимаю ее цель и не могу предложить хорошее имя, то использую имя `TODO`. Позднее в процессе рецензирования кода я обычно нахожу эти временные имена, и у меня будет больше желания (и больше оснований), чтобы вернуться и придумать более удачное имя вместо `TODO`.

Всем функциям нужны имена. Всем без исключений. Любое имя, пропущенное вами, усложняет чтение программы, усложняет ее отладку, расширение и последующее сопровождение.

## Стрелочные функции

Стрелочные функции всегда анонимны, даже если они (нечасто) используются способом, при котором им присваивается автоматически определенное имя. Я только что потратил несколько

страниц, объясняя, чем плохи анонимные функции; вероятно, вы догадываетесь, чем плохи стрелочные функции.

Не используйте их как общую замену для обычных функций. Да, они более компактны, но за эту компактность приходится расплачиваться ключевыми визуальными ограничителями, которые помогают нашему мозгу быстро разбирать читаемый текст. И в контексте нашего обсуждения они анонимны, что ухудшает их удобочитаемость.

У стрелочных функций есть свое предназначение, но оно не сводится к экономии нескольких нажатий клавиш. Стрелочные функции обладают поведением *лексического this*, что выходит за рамки материала книги.

Вкратце: стрелочные функции вообще не определяют идентификатор `this` как ключевое слово. Если вы используете `this` внутри стрелочной функции, это имя ведет себя точно так же, как любая другая ссылка на переменную, т. е. происходит поиск по цепочке областей видимости для нахождения области видимости функции (нестрелочной функции), где оно определяется, и использование найденной ссылки. Другими словами, стрелочные функции рассматривают `this` как любую другую лексическую переменную.

Если вы привыкли к трюкам типа `var self = this` или предпочитаете вызывать `.bind(this)` для внутренних функциональных выражений просто для того, чтобы унаследовать `this` от внешней функции, словно это лексическая переменная, то стрелочные функции `=>` однозначно будут лучшим вариантом. Они проектировались специально для решения этой проблемы.

Таким образом, в редких случаях, когда вам требуется поведение лексического `this`, используйте стрелочную функцию. Это лучший инструмент для решения вашей задачи. Но помните, что при этом вы соглашаетесь на все недостатки анонимных функций. Вам придется потратить дополнительные усилия, чтобы компенсировать снижение удобочитаемости — например, добавить в код содержательные имена переменных и комментарии.

## Разновидности IIFE

У всех функций должны быть имена. Я ведь уже говорил это, да? Это утверждение распространяется и на IIFE:

```
(function(){
    // не делайте так!
})();

(function doThisInstead(){
    // ..
})();
```

Как выбрать имя для IIFE? Определите, для чего существует IIFE. Для чего вам нужна область видимости в этой точке программы? Вы скрываете переменную-кэш для записей с информацией о студентах?

```
var getStudents = (function StoreStudentRecords(){
    var studentRecords = [];

    return function getStudents() {
        // ..
    }
})();
```

Я присвоил IIFE имя `StoreStudentRecords`, потому что оно описывает предназначение IIFE: хранение записей студентов. У каждого выражения IIFE должно быть имя. Без исключений.

IIFE обычно определяются заключением функционального выражения в круглые скобки `(..)`, как показано в предыдущих фрагментах. Тем не менее это не единственный способ определения IIFE. С технической точки зрения первый окружающий набор скобок `(..)` используется только по одной причине: чтобы ключевое слово `function` не находилось в позиции, в которой оно могло бы рассматриваться парсером JS как объявление функции. Но есть и другие синтаксические способы, предотвращающие его разбор как объявления:

```
!function thisIsAnIIFE(){
    // ..
}();

+function soIsThisOne(){
    // ..
}();

~function andThisOneToo(){
    // ..
}();
```

!, +, ~ и несколько других унарных операторов (операторов с одним операндом) можно поставить перед `function`, чтобы преобразовать объявление в выражение. Тогда последний вызов `()` становится допустимым и превращается в IIFE.

Лично мне нравится использовать унарный оператор `void` при определении автономных IIFE:

```
void function yepItsAnIIFE() {
    // ..
}();
```

Преимущество `void` в том, что этот оператор четко сообщает в начале функции, что IIFE не возвращает никакого значения.

Как бы вы ни определяли свои IIFE, проявите к ним уважение и присвойте имена.

## Поднятие: функции и переменные

В главе 5 было описано как *поднятие функций*, так и *поднятие переменных*. Так как поднятие (hoisting) часто характеризуется как ошибка при проектировании JS, я хочу кратко объяснить, почему обе формы поднятия *могут* быть полезными и ими не стоит пренебрегать. Чтобы рассмотреть поднятие на более глубоком уровне, вспомните его основные преимущества:

- сначала идет исполняемый код, потом объявления функций;
- семантическое размещение объявлений переменных.

## Поднятие функций

Вспомните, что следующая программа работает благодаря *поднятию функций*:

```
getStudents();
// ..

function getStudents() {
    // ..
}
```

Объявление `function` поднимается в процессе компиляции; это означает, что `getStudents` — идентификатор, объявляемый для всей области видимости. Кроме того, идентификатор `getStudents` автоматически инициализируется ссылкой на функцию (также в начале области видимости).

Почему это полезно? Я использую поднятие функций из-за того, что исполняемый код размещается в любой области видимости наверху, а все дальнейшие объявления (функции) — внизу. Это позволяет мне легко найти код, который будет выполняться в любой заданной области, вместо того чтобы прокручивать и прокручивать страницы в поисках закрывающей фигурной скобки }, отмечающей конец области видимости/функции.

Я пользуюсь преимуществами этого *обратного позиционирования* на всех уровнях области видимости:

```
getStudents();

// *************

function getStudents() {
    var whatever = doSomething();

    // и т. д.

    return whatever;
    // *************
```

```
    function doSomething() {
        // ..
    }
}
```

Когда я открываю такой файл, самая первая строка содержит исполняемый код, который активизирует его поведение. Найти эту строку несложно, она хорошо видна. Если мне в дальнейшем понадобится найти и просмотреть функцию `getStudents()`, ее первая строка также содержит исполняемый код. И только если понадобится просмотреть подробности того, что делает `doSomething()`, я начинаю искать определение функции внизу.

Иначе говоря, я думаю, что *поднятие функций* упрощает чтение кода за счет более естественного и последовательного порядка — от начала к концу.

## Поднятие переменных

Как насчет *поднятия переменных*?

Несмотря на то что объявления `let` и `const` поднимаются, эти переменные не могут использоваться в своих зонах TDZ (см. главу 5). Таким образом, следующее обсуждение относится только к объявлениям `var`. Прежде чем продолжать, я должен признать: почти во всех случаях я полностью согласен с тем, что *поднятие переменных* — неудачная идея:

```
pleaseDontDoThis = "bad idea";
    // much later
var pleaseDontDoThis;
```

Хотя подобное обратное позиционирование было полезно для *поднятия функций*, в данном случае я считаю, что оно обычно усложняет анализ кода.

Однако существует одно исключение, которое встречается — хотя и редко — в моих программах. Оно связано с тем, где я размещаю объявления `var` в определениях модулей CommonJS.

Типичная структура определения модулей в Node выглядит так:

```
// зависимости
var aModuleINeed = require("very-helpful");
var anotherModule = require("kinda-helpful");

// открытый API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    // ..
});

// ******************************
// приватная реализация

var cache = { };
var otherData = [ ];

function getStudents() {
    private implementation// ..
}

function addStudents() {
    // ..
}
```

Обратите внимание: переменные `cache` и `otherData` находятся в «приватном» разделе структуры модуля. Дело в том, что я не планирую предоставлять их для открытого доступа. По этой причине я выбираю такую структуру модуля, чтобы они находились вместе с другими скрытыми подробностями реализации модуля.

Но я видел несколько редких случаев, в которых мне требовалось выполнить присваивание этих значений *выше*, до объявления экспортируемого открытого API модуля. Пример:

```
// открытый API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null,cache)
});
```

Было нужно, чтобы переменной `cache` уже было присвоено значение, потому что это значение используется при инициализации открытого API (частичное применение `the .bind(..)`).

Следует ли переместить команду `var cache = { .. }` наверх, перед инициализацией открытого API? Возможно. Но тогда становится менее очевидно, что `var cache` является подробностью *приватной* реализации. Компромиссное решение, которое я применял (достаточно редко):

```
cache = {}; // используется здесь, но объявляется внизу

// открытый API
var publicAPI = Object.assign(module.exports,{
    getStudents,
    addStudents,
    refreshData: refreshData.bind(null,cache)
});

// ******************************
// приватная реализация

var cache /* = {}*/;
```

Видите эффект *поднятия переменной*? Я объявил переменную `cache` там, где ей место (логически), но в этом редком случае я воспользовался ей выше, в той области, где нужно, чтобы она была инициализирована. Я даже оставил подсказку о значении, которое присваивается `cache`, в комментарии.

И это буквально единственный случай, в котором я использую поднятие переменной для присваивания значения, предшествующего объявлению переменной. Но я считаю, что это разумное исключение, которым следует пользоваться осторожно.

## В защиту var

Раз уж речь зашла о *поднятии переменных*, поговорим немного о `var` — любимом «злодее» многих разработчиков, которого они обвиняют во многих грехах. В главе 5 мы изучили объявления

`let`/`const` и я пообещал вернуться к теме и показать, какое место `var` занимает в общей картине.

И пока я буду приводить аргументы, помните:

- объявления `var` никогда не были сломаны;
- `let` — ваш друг;
- полезность `const` ограниченна;
- используйте лучшее из обоих миров: `var` *и* `let`.

### Не отказывайтесь от var

С объявлениями `var` все нормально, и они прекрасно работают. Они существовали около 25 лет и будут жить еще столько же. Утверждения о том, что объявления `var` сломаны, устарели, опасны или плохо спроектированы, — массовое заблуждение. Означает ли это, что `var` идеально подходит для каждого объявления в вашей программе? Конечно, нет. Но место в ваших программах для него все же найдется. Отказываться от его использования только из-за того, что кто-то в вашей команде выбрал агрессивный режим синтаксического анализатора, который поперхнулся на `var`, — значит действовать по принципу «назло маме уши отморожу».

А теперь, когда вы разозлились, объясню свою позицию.

Для протокола: я поклонник `let` для объявлений с блоковой областью видимости. Мне не нравится TDZ, и я думаю, что это было ошибкой. Но объявления `let` сами по себе превосходны. Я часто пользуюсь ими. Наверное, пользуюсь ими не реже, а то и чаще, чем `var`.

### Постоянная путаница с const

С другой стороны, ключевым словом `const` я пользуюсь не так часто. Не буду углубляться в подробные объяснения, почему это так, но все сводится к тому, что использование `const` *просто не стоит того*. Другими словами, хотя в некоторых случаях `const` может принести определенную пользу, эти преимущества компенсируются много-

летними неприятностями, связанными с неправильным пониманием `const` в разных языках еще задолго до его включения в JS.

Многие считают, что `const` создает значения, которые не могут изменяться (ошибочное представление, в высшей степени распространенное в сообществах разработчиков во многих языках), тогда как в действительности оно всего лишь запрещает повторное присваивание.

```
const studentIDs = [ 14, 73, 112 ];

// позднее
 studentIDs.push(6); // стоп, погодите... что?!
```

Используя `const` с изменяемым значением (таким как массив или объект), вы расставляете ловушку будущему разработчику (или читателю) вашего кода. Дело в том, что *неизменяемость значений* — совсем не то же самое, что *неизменяемость по присваиванию*.

Я просто считаю, что не следует расставлять эти ловушки. Сам я использую `const` только в одной ситуации — когда присваивается уже неизменяемое значение (например, `42` или `"Hello, friends!"`) и когда идентификатор является константой в том смысле, что он становится именованным заместителем для литерального значения для семантических целей. Именно для таких применений `const` подходит лучше всего. Впрочем, в моем коде такие случаи встречаются относительно редко.

Если бы повторное присваивание переменным было серьезной проблемой, то использование `const` было бы более полезным. Однако повторное присваивание не играет заметной роли в причине ошибок. Список факторов, которые могут вызвать ошибки в программах, достаточно длинный, но случайное повторное присваивание находится далеко не в первых позициях этого списка.

Объединим это с тем фактом, что `const` (и `let`) предназначено для использования в блоках, а блоки должны быть короткими, так что сама область, в которой могут применяться объявления `const`, весьма невелика. `const` в строке 1 нашего 10-строчного блока только говорит вам что-то о следующих девяти строках. Причем

то, что оно вам говорит, становится и так очевидно при взгляде на эти девять строк: переменная никогда не находится слева от `=`; ей нигде не присваивается значение повторно.

И это все, что реально делает `const`. В остальном оно бесполезно. Если сравнить с серьезной путаницей с неизменяемостью значений/неизменяемостью по присваиванию, `const` в значительной мере теряет часть своей привлекательности.

Объявление `let` (или `var`!), которой нигде не присваивается значение повторно, уже является «константой» с точки зрения поведения, хотя эта «константность» не имеет гарантий компилятора. В большинстве случаев этого вполне достаточно.

### var и let

В моем представлении `const` редко приносит пользу, так что в гонке фактически участвуют только `let` и `var`. Впрочем, это вряд ли можно назвать гонкой, потому что единственного победителя не будет. Выиграть могут сразу двое... каждый в своем классе.

Дело в том, что вам в своих программах стоит использовать как `var`, так и `let`. Эти ключевые слова не являются взаимозаменяемыми; не используйте `var` там, где ситуация требует `let`, но также и не стоит использовать `let` там, где наиболее уместным будет `var`.

Где же использовать ключевое слово `var`? В каких обстоятельствах оно подойдет лучше, чем `let`?

Прежде всего, я всегда использую `var` в области видимости верхнего уровня любой функции независимо от того, находится ли оно в начале, середине или конце функции. Также я использую `var` в глобальной области видимости, хотя и стараюсь свети использование глобальной области видимости к минимуму.

Зачем использовать `var` для функциональной области видимости? Потому что `var` делает именно это. Для задачи определения функциональной области видимости для объявления буквально не существует лучшего инструмента, чем тот, который делал именно это в течение 25 лет.

Вы можете использовать `let` в этой области видимости верхнего уровня, но это не лучший инструмент для такой задачи. Я также обнаружил, что если вы будете использовать `let` повсюду, будет менее очевидно, какие объявления были спроектированы как локализованные, а какие должны использоваться по всей функции.

С другой стороны, я редко использую `var` внутри блока. Для этого существует `let`. Используйте наиболее подходящий инструмент для задачи. Если вы видите ключевое слово `let`, оно сообщает, что вы имеете дело с локализованным объявлением. Если вы видите ключевое слово `var`, оно сообщает, что вы имеете дело с объявлением верхнего уровня функции. В общем, все просто.

```
function getStudents(data) {
    var studentRecords = [];

    for (let record of data.records) {

        let id = `student-${ record.id }`;
        studentRecords.push({
            id,
            record.name
        });
    }

    return studentRecords;
}
```

Переменная `studentRecords` предназначена для использования в границах всей функции. `var` — лучший инструмент, который сообщит читателю кода эту информацию. С другой стороны, `record` и `id` предназначены для использования только в более узкой области видимости итерации цикла, поэтому для этой цели лучше всего подойдет `let`.

Помимо семантического аргумента о *наиболее подходящем инструменте* ключевое слово `var` также обладает рядом характеристик, которые в некоторых ограниченных обстоятельствах делают его более мощным.

Один из примеров — когда переменная используется исключительно в цикле, но его условная секция не видит объявлений блоковой видимости внутри итерации:

```
function commitAction() {
    do {
        let result = commit();
        var done = result && result.code == 1;
    } while (!done);
}
```

Здесь переменная `result` очевидно используется только внутри блока, поэтому мы выбираем `let`. Но переменная `done` — другое дело. Она полезна только в границах цикла, но секция `while` не видит объявления `let`, расположенные внутри цикла. По этой причине мы идем на компромисс и используем `var`, чтобы переменная `done` поднималась во внешнюю область видимости и была видна в ней.

Альтернатива — объявление `done` за пределами цикла — отделяет ее от места первого использования и заставляет вас либо выбрать значение по умолчанию, либо, что еще хуже, оставить переменную без присваивания и создать неоднозначность для читателя кода. Я думаю, что решение с `var` внутри цикла здесь будет предпочтительным.

Другая полезная характеристика `var` проявляется в объявлениях внутри непреднамеренных блоков. Непреднамеренными называются блоки, которые создаются, потому что синтаксису требуется блок, но разработчик на самом деле не собирался создавать локализованную область видимости. Лучшей иллюстрацией непреднамеренных областей видимости служит команда `try..catch`:

```
function getStudents() {
    try {
        // не является блоковой областью видимости
        var records = fromCache("students");
    }
    catch (err) {
        // возвращаемся к значению по умолчанию
```

```
        var records = [];
    }
    // ..
}
```

Да, есть и другие способы структурирования этого кода. Но думаю, что это *лучший* вариант с учетом всех плюсов и минусов.

Я не хочу объявлять переменную `records` (с `var` или `let`) за пределами блока `try`, а затем присваивать ей значение в одном или обоих блоках. Я предпочитаю, чтобы исходные объявления были как можно ближе (в идеале в одной строке) к первому использованию переменной. В этом простом примере расстояние может составлять всего пару строк, но в реальном коде таких строк может быть намного больше. Чем больше разрыв, тем сложнее определить, какой переменной из какой области видимости присваивается значение. Ключевое слово `var`, используемое при фактическом присваивании, делает его более однозначным.

Также обратите внимание на то, что я использовал `var` в обоих блоках, `try` и `catch`. Этим я хотел дать сигнал читателю кода, что переменная `records` будет объявлена при любом выборе пути. Технически это решение работает, потому что `var` поднимается только один раз в область видимости функции. Тем не менее это хороший семантический сигнал, который напомнит читателю кода, что обеспечивает каждое из объявлений `var`. Если бы ключевое слово `var` использовалось только в одном из блоков, а вы бы прочитали только другой блок, то не смогли бы так же легко понять, откуда взялась переменная `records`.

И в этом, как мне кажется, заключается маленькая суперспособность `var`. Это объявление может не только обходить непреднамеренные блоки `try..catch`, но и многократно встречаться в области видимости функции. С `let` такое сделать невозможно. В этом нет ничего плохого, это маленькая полезная возможность. Рассматривайте `var` как декларативную аннотацию, которая при каждом использовании напоминает вам, откуда взялась переменная. «Ага, правильно, она принадлежит всей функции».

Эта суперспособность повторяемой аннотации может пригодиться и в других случаях:

```
function getStudents() {
    var data = [];

    // обработка данных
    // .. еще 50 строк кода ..

    // аннотация для напоминания
    var data;

    // повторное использование данных
    // ..
}
```

Вторая конструкция `var data` не является переобъявлением `data`; она всего лишь указывает читателю кода, что `data` объявляется на уровне функции. Благодаря этому читателю не придется прокручивать код вверх на 50+ строк, чтобы найти исходное объявление.

Я абсолютно нормально отношусь к повторному использованию переменных для нескольких целей в области видимости функции. Я также абсолютно нормально отношусь к двум сценариям использования переменных, разделенных лишь несколькими строками кода. В обоих случаях возможность безопасного повторного объявления (а на самом деле аннотации) командой `var helps` поможет всегда знать, откуда взялись мои данные, независимо от того, где я нахожусь в функции.

И снова `let`, как ни печально, сделать этого не сможет.

Также есть другие нюансы и сценарии, в которых `var` приносит некоторую пользу, но я не собираюсь развивать эту тему. А значит, `var` может пригодиться в наших программах наряду с `let` (и отдельных вкраплений `const`). Вы же хотите творчески использовать инструменты, предоставляемые языком JS, чтобы донести более содержательную информацию для читателей вашего кода?

Не отказывайтесь от такого полезного инструмента, как `var`, только потому, что кто-то пристыдил вас и заставил думать, что это уже не круто. Не избегайте объявлений `var` из-за того, что когда-то вы

на них споткнулись. Изучите эти инструменты и используйте каждый из них для тех задач, для которых они лучше подходят.

## Для чего была создана TDZ?

Концепция TDZ (Temporal Dead Zone) объяснялась в главе 5. Я показал, как она возникает в программе, но не объяснил, почему было необходимо вводить ее в JS. Кратко рассмотрим основные причины для создания TDZ.

Некоторые ключевые моменты в истории происхождения TDZ:

- `const` не могут изменяться;
- все дело во времени;
- должны ли объявления `let` быть больше похожими на `const` или `var`?

### С чего все началось

На самом деле область TDZ появилась из-за `const`.

В начале разработки ES6 комитету TC39 пришлось решать, должны ли `const` (и `let`) подниматься в начало своих блоков. Было решено, что эти объявления должны подниматься по аналогии с тем, как это происходит с `var`. В противном случае, я думаю, что опасения были отчасти обусловлены возможной путаницей из-за замещения в середине области видимости:

```
let greeting = "Hi!";

{
    // что здесь должно выводиться?
    console.log(greeting);

    // .. много строк кода ..

    // Затенение переменной `greeting`
    let greeting = "Hello, friends!";

    // ..
}
```

Что делать с командой `console.log(..)`? Если команда просто выведет «Hi!», будет ли это разумно с точки зрения разработчика JS? Получается, что затенение начинает работать только во второй половине блока, а в первой не работает — это выглядит довольно странно. Такое поведение не интуитивно и не похоже на JS. Соответственно, объявления `let` и `const` должны подниматься в начало блока и быть видимыми повсюду.

Но если `let` и `const` поднимаются в начало блока (подобно тому как `var` поднимается в начало функции), почему бы не разрешить `let` и `const` автоматически инициализироваться (значением `undefined`) по аналогии с `var`? Следующий пример демонстрирует основную проблему:

```
{
    // что здесь должно выводиться?
    console.log(studentName);

    // позднее

    const studentName = "Frank";

    // ..
}
```

Представьте, что `studentName` не только поднимается в начало блока, но и автоматически инициализируется значением `undefined`. В первой половине блока переменная `studentName` будет иметь наблюдаемое значение `undefined` (это относится к команде `console.log(..)`). После достижения команды `const studentName = ..` переменной `studentName` теперь присваивается значение `"Frank"`. С этой точки и далее переменной `studentName` уже невозможно присвоить новое значение.

Но разве не будет странно и удивительно выглядеть то, что константа наблюдаемо имеет два разных значения — сначала `undefined`, затем `"Frank"`? Это вроде бы противоречит нашим представлениям о смысле константы — она должна наблюдаться только с одним значением.

Итак... возникает проблема. Мы не можем автоматически инициализировать `studentName` значением `undefined` (или любым другим, если на то пошло).

Но переменная должна существовать во всей области видимости. Что делать с периодом от того момента, когда она начинает существовать (начало области видимости), и до того, как ей будет присвоено значение?

Мы называем этот период времени мертвой зоной, или TDZ (Temporal Dead Zone). Для предотвращения путаницы было определено, что любые обращения к переменной, находящейся в области TDZ, недопустимы и должны приводить к ошибке TDZ.

Окей, признаю — эта цепочка рассуждений выглядит разумно.

### TDZ и let

Но пока речь шла только о `const`. А как насчет `let`?

Комитет TC39 принял решение: так как нам нужны TDZ для `const`, то можно предусмотреть TDZ и для `let`. *Собственно, добавляя TDZ для let, мы тем самым отбиваем у людей охоту использовать уродливое поднятие переменных*. Так что здесь выбор объяснялся логической целостностью и, возможно, желанием добавить некоторую долю социотехники, чтобы повлиять на поведение разработчиков.

Мои возражения: если вы хотите логической целостности, то равняйтесь на `var`, а не на `const`; у `let` определенно больше общего с `var`, чем с `const`. Это особенно справедливо, потому что согласованность с `var` уже была выбрана для всей концепции поднятия в начало области видимости. Пусть `const` будет особым случаем с TDZ, и пусть проблема с TDZ решается легко: просто избегайте TDZ, всегда объявляя свои константы в начале области видимости. На мой взгляд, это было бы более разумно.

Но, к сожалению, вышло все не так. `let` имеет TDZ, потому что TDZ необходима для `const`, из-за того что `let` и `const` следуют

примеру `var` с поднятием в начало (блоковой) области видимости. Голова идет кругом? Перечитайте несколько раз.

## Остаются ли синхронные обратные вызовы замыканиями?

В главе 7 были представлены два разных модуля для осмысления замыканий:

- замыкание рассматривается как экземпляр функции, запоминающей свои внешние переменные, даже если эта функция передается и **вызывается в** других областях видимости;
- замыкание рассматривается как экземпляр функции, окружение области видимости которого сохраняется на месте, а ссылки на эту функцию передаются и **вызываются из** других областей видимости.

Эти модели не так уж сильно отличаются, но они рассматривают происходящее с разных точек зрения. И эти точки зрения изменяют то, что мы определяем как замыкание.

Не заплутайте на этой извилистой тропе со всеми ее замыканиями и обратными вызовами:

- обратный вызов чего (или где)?
- возможно, термин «синхронный обратный вызов» выбран неудачно;
- функции *IIF* не перемещаются, зачем им замыкание?
- временная задержка — ключевой фактор для замыканий.

### Что такое обратный вызов?

Прежде чем возвращаться к замыканиям, потрачу еще немного времени на объяснение термина «обратный вызов» (callback). Обычно считается, что обратный вызов включает как асинхронные, так

и синхронные обратные вызовы. На мой взгляд, это не лучшая идея. Хочу объяснить, почему я так считаю, и предложить другой термин.

Начнем с *асинхронного обратного вызова* — ссылки на функцию, которая будет вызвана *позднее*. Что означает обратный вызов в этом случае?

Он означает, что текущий код завершился, сделал паузу или приостановил свое выполнение и что при последующем вызове этой функции управление передается приостановленной программе, что приводит к возобновлению ее выполнения. А конкретнее, точкой повторного входа становится код, «завернутый» в ссылку на функцию:

```
setTimeout(function waitForASecond(){
    // сюда JS должен обратиться с обратным вызовом
    // при истечении таймера
},1000);

// здесь текущая программа завершается
// или приостанавливается
```

В этом контексте термин «обратный вызов» выглядит разумно. Движок JS возобновляет приостановленную программу, обращаясь с обратным вызовом в конкретную точку. Хорошо, обратный вызов в данном случае работает асинхронно.

## Синхронный обратный вызов?

Но как насчет *синхронных обратных вызовов*? Пример:

```
function getLabels(studentIDs) {
    return studentIDs.map(
        function formatIDLabel(id){
            return `Student ID: ${
                String(id).padStart(6)
            }`;
        }
    );
}
```

```
getLabels([ 14, 73, 112, 6 ]);
// [
// "Student ID: 000014",
// "Student ID: 000073",
// "Student ID: 000112",
// "Student ID: 000006"
// ]
```

Стоит ли называть `formatIDLabel(..)` обратным вызовом? Действительно ли функция `map(..)` обращается с обратным вызовом к программе, вызывая предоставленную нами функцию?

Здесь нет никакого обратного вызова как такового, потому что программа не приостанавливается и не завершается. Мы передаем функцию (вернее, ссылку) из одной части программы в другую, после чего немедленно вызываем ее.

Есть и другие устоявшиеся термины, которые могут подойти для решаемой задачи — передача функции (вернее, ссылки), чтобы другая часть программы могла вызвать ее по вашему поручению. Происходящее можно рассматривать как *внедрение зависимостей* (DI, Dependency Injection) или *инверсию управления* (IoC, Inversion of Control).

DI можно охарактеризовать как передачу необходимых частей функциональности другим частям программы, чтобы их можно было вызывать для завершения их операций. Это описание неплохо подходит для вызова `map(..)`, не правда ли? Вспомогательная функция `map(..)` умеет перебирать значения в списке, но не знает, что делать с этими значениями. Именно поэтому мы передаем ей функцию `formatIDLabel(..)`. Мы передаем *зависимость*.

IoC — довольно похожая взаимосвязанная концепция. Инверсия управления означает, что происходящее в программе не находится под управлением текущей области программы — вы передаете управление другой части программы. Мы завернули логику вычисления строки метки в функцию `formatIDLabel(..)`, а затем передали управление ее вызовом функции `map(..)`.

Стоит заметить, что Мартин Фаулер (Martin Fowler) приводит IoC как признак, по которому можно отличить фреймворк от

библиотеки: вы вызываете функции библиотеки, а фреймворк вызывает ваши функции[1].

В контексте нашего обсуждения как DI, так и IoC может послужить альтернативным термином для *синхронных обратных вызовов*.

Однако у меня есть другое предложение. Будем называть *синхронные обратные вызовы* (вернее, функции, которые мы так ранее называли) *IIF* (Inter-Invoked Functions, т. е. *функции опосредованного вызова*). Да, конечно, я здесь обыгрываю термин IIFE. Такие виды функций вызываются *опосредованно*, т. е. они вызываются другой сущностью (в отличие от IIFE, которые вызывают сами себя непосредственно).

Какими же отношениями связываются асинхронные обратные вызовы с IIF? *Асинхронный обратный вызов* — это функция IIF, которая вызывается асинхронно, а не синхронно.

## Синхронные замыкания?

Итак, мы присвоили синхронным обратным вызовам новое название IIF, и мы можем вернуться к главному вопросу: являются ли IIF примером замыкания? Очевидно, IIF должна обращаться к переменным из внешней области видимости, чтобы она могла считаться замыканием. IIF `formatIDLabel(..)` из предыдущего примера не обращается ни к каким переменным вне своей области видимости, поэтому она определенно не является замыканием.

Как насчет IIF с внешними ссылками? Являются ли они замыканиями?

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");

    labels.forEach(
        function renderLabel(label){
            var li = document.createELement("li");
```

[1] Inversion of Control, Martin Fowler, https://martinfowler.com/bliki/InversionOfControl.html, 26 июня 2005 г.

```
            li.innerText = label;
            list.appendChild(li);
        }
    );
}
```

Внутренняя IIF `renderLabel(..)` обращается к `list` из окружающей области видимости, поэтому в данном случае IIF может иметь замыкание. Но здесь начинают играть важную роль определение/модель, выбранные нами для замыкания.

- Если `renderLabel(..)` является функцией, которая передается в другую точку, и эта функция затем вызывается, тогда `renderLabel(..)` действительно проявляет замыкание, потому что замыкание должно сохранять свой доступ к исходной цепочке областей видимости.
- Но если, как описано в альтернативной концептуальной модели из главы 7, `renderLabel(..)` остается на месте, и `forEach(..)` передается только ссылка, есть ли какая-то необходимость в том, чтобы замыкание сохраняло цепочку областей видимости `renderLabel(..)`, пока она выполняется синхронно прямо внутри своей области видимости?

Нет. Это обычная лексическая область видимости.

Чтобы понять, почему это происходит, рассмотрим альтернативную форму `printLabels(..)`:

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");

    for (let label of labels) {
        // просто обычный вызов функции в ее собственной области
        // видимости, правильно? Замыканием не является!
        renderLabel(label);
    }

    // **************

    function renderLabel(label) {
        var li = document.createELement("li");
```

```
        li.innerText = label;
        list.appendChild(li);
    }
}
```

Эти две версии `printLabels(..)` фактически одинаковы.

Вторая версия определенно не является примером исключения, по крайней мере ни в каком полезном или наблюдаемом смысле. Это всего лишь лексическая видимость. Первая версия, в которой `forEach(..)` вызывает нашу функцию, фактически делает то же самое. Она тоже является не замыканием, а простой функцией в лексической области видимости.

## Временная задержка

Кстати, в главе 7 кратко упоминались механизмы частичного применения и каррирования (зависящие от замыкания). Это интересный сценарий, в котором может применяться ручное каррирование:

```
function printLabels(labels) {
    var list = document.getElementByID("labelsList");
    var renderLabel = renderTo(list);

    // definitely closure this time!
    labels.forEach( renderLabel(label) );

    // **************

    function renderTo(list) {
        return function createLabel(label){
            var li = document.createELement("li");
            li.innerText = label;
            list.appendChild(li);
        };
    }
}
```

Внутренняя функция `createLabel(..)`, которую мы присваиваем `renderLabels`, замыкается на `list`, так что здесь определенно используется замыкание.

Замыкание позволяет нам запомнить `list` на будущее, тогда как выполнение фактической логики создания меток откладывается от вызова `renderTo(..)` до последующих вызовов IIF `createLabel(..)` из `forEach(..)`. Возможно, это будет лишь совсем короткий момент, но теоретически продолжительность может быть любой.

## Вариации на тему классических модулей

В главе 8 рассматривался паттерн «Классический модуль», который выглядит примерно так:

```
var StudentList = (function defineModule(Student){
    var elems = [];

    var publicAPI = {
        renderList() {
            // ..
        }
    };

    return publicAPI;

})(Student);
```

Следует заметить, что мы передаем `Student` (другой экземпляр модуля) как зависимость. Однако у этой формы модуля существует много полезных модификаций. Следующие подсказки помогут вам распознать эти вариации:

- знает ли модуль о своем API?
- даже если мы используем изощренный загрузчик модуля, это всего лишь классический модуль;
- некоторые модули должны работать универсально.

## Где мой API?

Прежде всего, большинство классических модулей не определяет и не использует `publicAPI` так, как я показал в этом коде. Вместо этого они выглядят примерно так:

```
var StudentList = (function defineModule(Student){
    var elems = [];

    return {
        renderList() {
            // ..
        }
    };

})(Student);
```

Этот пример отличается только одним: в нем напрямую возвращается объект, который служит открытым API для модуля, вместо предварительного сохранения его во внутренней переменной `publicAPI`. Так определяется подавляющее большинство классических модулей.

Но я безусловно предпочитаю и всегда использую сам первую форму с `publicAPI`. Это объясняется двумя причинами:

- `publicAPI` — семантический дескриптор, который упрощает чтение программы, наглядно поясняя предназначение объекта;
- внутренняя переменная `publicAPI`, которая ссылается на тот же объект внешнего открытого API, может быть полезной, если вам потребуется обратиться или изменить API на протяжении срока жизни модуля.

Например, вы можете захотеть вызвать одну из общедоступных функций внутри модуля. А может быть, вы захотите добавить или удалить методы в зависимости от некоторых условий или обновить предоставляемое свойство. В любом случае *отказ* от хранения ссылки для обращения к собственному API выглядит довольно странно. Не так ли?

### Асинхронное определение модуля (AMD)

Другая разновидность классических модулей — модули в стиле AMD (которые были популярны несколько лет назад) вроде тех, которые поддерживаются RequireJS:

```
define([ "./Student" ],function StudentList(Student){
    var elems = [];

    return {
        renderList() {
            // ..
        }
    };
});
```

Если внимательно присмотреться к `StudentList(..)`, мы видим, что это фабричная функция классического модуля. Внутренние механизмы `define(..)` (предоставляется RequireJS) выполняют функцию `StudentList(..)`, передавая ей любые экземпляры модулей, объявленные как зависимости. Возвращаемым значением является объект, представляющий открытый API для модуля.

Все происходящее базируется на абсолютно тех же принципах (включая работу замыканий), которые были описаны для классических модулей.

## Универсальные модули (UMD)

Последняя разновидность модулей, которую мы рассмотрим — UMD, — является не столько конкретным точным форматом, сколько коллекцией очень похожих форматов. Она была спроектирована для улучшения взаимодействия (без преобразований средствами сборки) для модулей, которые могут загружаться в браузерах загрузчиками в стиле AMD или в Node. Лично я все еще публикую многие из своих вспомогательных библиотек в форме UMD.

Типичная структура UMD выглядит так:

```
(function UMD(name,context,definition){
    // загружается загрузчиком в стиле AMD?
    if (
        typeof define === "function" &&
        define.amd
    ) {
        define(definition);
    }
    // В Node?
    else if (
        typeof module !== "undefined" &&
        module.exports
    ) {
        module.exports = definition(name,context);
    }
    // предполагается автономный сценарий для браузера
    else {
        context[name] = definition(name,context);
    }
})("StudentList",this,function DEF(name,context){

    var elems = [];

    return {
        renderList() {
            // ..
        }
    };

});
```

И хотя структура выглядит немного необычно, UMD в действительности является выражением IIFE.

Отличается здесь то, что главная часть функционального выражения (наверху) IIFE содержит серию команд `if..else` для определения одной из трех поддерживаемых сред, в которой загружается модуль.

В последних круглых скобках `()`, которые обычно вызывают IIFE, передаются три аргумента: `"StudentsList"`, `this` и другое функцио-

нальное выражение. Сопоставив эти аргументы со своими параметрами, вы увидите, что это `name`, `context` и `definition` соответственно. `"StudentList"` (`name`) содержит имя модуля, в основном на случай его определения как глобальной переменной. `this` (`context`) обычно соответствует `window` (т. е. глобальному объекту; см. главу 4) для определения модуля по имени. `definition(..)` вызывается для фактического получения определения модуля, и можно заметить, что это просто классическая форма модуля!

Невозможно отрицать, что на момент написания книги модули ESM(ES Modules) набирают популярность и стремительно распространяются. Но за последние 20 лет были написаны миллионы модулей, и во всех использовались те или иные разновидности классических модулей, предшествовавшие ESM, поэтому все еще очень важно уметь читать и понимать их.

# Б Практика

В этом приложении приводятся некоторые непростые и интересные упражнения, которые должны проверить и укрепить ваше понимание основных тем книги. Очень полезно попытаться выполнить упражнения самостоятельно — в настоящем редакторе кода — вместо того, чтобы переходить к решениям в конце. Не читерите!

Эти упражнения не имеют единственно правильного ответа. Ваш подход может незначительно (или значительно) отличаться от представленных решений, и это нормально.

Не думайте, что эти упражнения оценивают ваши навыки программирования. Надеюсь, что вы закроете эту книгу, чувствуя большую уверенность в том, что справитесь с этими задачами, благодаря прочному фундаменту из полученных знаний. И это единственная цель этого приложения. Если вы довольны своим кодом, то им доволен и я!

## Камешки и банки

Помните рис. 2 из главы 2?

```
var students = [
    { id: 14, name: "Kyle" },
    { id: 73, name: "Suzy" },
    { id: 112, name: "Frank" },
    { id: 6, name: "Sarah" }
];

function getStudentName(studentID) {
    for (let student of students) {
        if (student.id == studentID) {
            return student.name;
        }
    }
}

var nextStudent = getStudentName(73);

console.log(nextStudent);
// "Suzy"
```

**Рис. 2 (*глава 2*).** Границы областей видимости

В этом упражнении вам будет предложено написать любую программу, содержащую вложенные функции и блоковые области видимости, удовлетворяющую следующим ограничениям:

- если раскрасить все области видимости (включая глобальную) в разные цвета, понадобится не менее 6 цветов. Обязательно добавьте комментарий, помечающий каждую область видимости своим цветом.

  ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: найдите все предполагаемые области видимости, которые могут присутствовать в вашей программе;

- каждая область видимости содержит хотя бы один идентификатор;

- содержит по крайней мере две функциональные области видимости и по крайней мере две блоковые области видимости;
- по крайней мере одна переменная из внешней области видимости должна замещаться переменной из вложенной области видимости (см. главу 3);
- по крайней мере одна ссылка на переменную должна разрешаться в объявление переменной, находящейся не менее чем двумя уровнями выше в цепочке областей видимости.

Вы *можете* написать для этого упражнения мусорный код в стиле foo/bar/baz, но рекомендую попробовать написать нетривиальный реальный код, который хотя бы пытается делать нечто разумное.

Попробуйте выполнить упражнение самостоятельно, а затем сверьтесь с предлагаемым решением в конце этого приложения.

## Замыкания (часть 1)

Для начала потренируемся в использовании замыканий на примере некоторых математических операций, часто реализуемых на компьютерах: определения того, является ли значение простым (не делится нацело ни на какие числа, кроме 1 и самого себя), и генерирования списка простых множителей для заданного числа.

Пример:

```
isPrime(11); // true
isPrime(12); // false

factorize(11); // [ 11 ]
factorize(12); // [ 3, 2, 2 ] --> 3*2*2=12
```

Следующая реализация `isPrime(..)` адаптирована из библиотеки Math.js[1]:

[1] Math.js: isPrime(..), https://github.com/josdejong/mathjs/blob/develop/src/function/utils/isPrime.js, 3 марта 2020 года.

```
function isPrime(v) {
    if (v <= 3) {
        return v > 1;
    }
    if (v % 2 == 0 || v % 3 == 0) {
        return false;
    }
    var vSqrt = Math.sqrt(v);
    for (let i = 5; i <= vSqrt; i += 6) {
        if (v % i == 0 || v % (i + 2) == 0) {
            return false;
        }
    }
    return true;
}
```

Также приведу (довольно тривиальную) реализацию `factorize(..)` (не путайте с `factorial(..)` из главы 6):

```
function factorize(v) {
    if (!isPrime(v)) {
        let i = Math.floor(Math.sqrt(v));
        while (v % i != 0) {
            i--;
        }
        return [
            ...factorize(i),
            ...factorize(v / i)
        ];
    }
    return [v];
}
```

Я называю эту реализацию тривиальной, потому что она не оптимизирована по производительности. Она является бинарно-рекурсивной (т. е. непригодной для оптимизации хвостовых вызовов), а также создает множество промежуточных копий массива. Кроме того, найденные множители никак не упорядочиваются. Есть множество других алгоритмов для этой задачи, но для нашего упражнения я хотел использовать нечто короткое и достаточно понятное.

Если вы вызовете `isPrime(4327)` несколько раз в программе, то увидите, что каждый раз выполняются все десятки шагов сравнения/вычислений. Если присмотреться к функции `factorize(..)`, вы заметите, что она многократно вызывает `isPrime(..)` при вычислении списка множителей. Существует достаточно высокая вероятность того, что многие из этих вызовов повторяются. Слишком много работы пропадает даром!

Первая часть упражнения: воспользуйтесь замыканием для реализации кэша, в котором будут сохраняться результаты `isPrime(..)`, чтобы признак простоты заданного числа (`true` или `false`) вычислялся только один раз. Подсказка: подобное кэширование уже было продемонстрировано в главе 6 на примере `factorial(..)`.

Также можно заметить, что функция `factorize(..)` реализована с рекурсией, т. е. многократно вызывает саму себя. Это снова означает, что мы с большой вероятностью столкнемся с множеством повторных вызовов для вычисления простых множителей одного и того же числа. Во второй части упражнения вам предстоит реализовать тот же механизм кэширования из замыкания для `factorize(..)`.

Используйте разные замыкания для кэширования `isPrime(..)` и `factorize(..)`, вместо того чтобы размещать их в одной области видимости.

Попробуйте выполнить упражнение самостоятельно, а затем сверьтесь с предлагаемым решением в конце приложения.

## Несколько слов о памяти

Хочу сказать несколько слов о кэшировании из замыканий и об его последствиях для быстродействия приложения.

Нетрудно убедиться, что сохранение результатов повторяющихся вызовов повышает скорость вычислений (в отдельных случаях очень значительно). Но такое использование замыканий сопряжено с компромиссом, о котором вы должны очень хорошо знать.

И этот компромисс связан с использованием памяти. Фактически мы наращиваем объем нашего кэша (в памяти) неограниченно. Если функция будет вызвана много миллионов раз с уникальными входными данными, это приведет к серьезным затратам памяти. Затраты определенно могут быть оправданы, но только если вы ожидаете, что входные данные будут с большой вероятностью повторяться. Если практически каждый вызов будет иметь уникальные входные данные, а кэш почти никогда не будет использоваться, применять кэширование не стоит.

Также, возможно, стоит воспользоваться более сложным методом кэширования — например, кэшем LRU (Least Recently Used) с ограниченным размером; при заполнении кэша LRU из него вытесняются элементы с наиболее давним использованием.

Недостаток такого решения в том, что кэш LRU достаточно нетривиален сам по себе. Вам придется использовать высокооптимизированную реализацию LRU и четко представлять себе все плюсы и минусы, с которыми вы столкнетесь.

## Замыкания (часть 2)

В этом упражнении мы снова потренируемся в использовании замыканий. Для этого будет определена вспомогательная функция `toggle(..)`, которая возвращает функцию для перебора значений.

Вы передаете `toggle(..)` одно или несколько значений (аргументов), и получаете обратно функцию. Возвращенная функция поочередно возвращает значения из своего списка при повторных вызовах.

```
function toggle(/* .. */) {
    // ..
}

var hello = toggle("hello");
var onOff = toggle("on","off");
```

```
var speed = toggle("slow","medium","fast");

hello(); // "hello"
hello(); // "hello"

onOff(); // "on"
onOff(); // "off"
onOff(); // "on"

speed(); // "slow"
speed(); // "medium"
speed(); // "fast"
speed(); // "slow"
```

Граничный случай, при котором `toggle(..)` не передается ни одно значение, не особо важен; в этом случае возвращенный экземпляр функции может просто всегда возвращать `undefined`.

Попробуйте выполнить упражнение самостоятельно, а затем сверьтесь с предлагаемым решением в конце приложения.

## Замыкания (часть 3)

В третьем и последнем упражнении, посвященном замыканиям, будет реализован простейший калькулятор. Функция `calculator()` создаст экземпляр калькулятора, поддерживающего свое состояние, в форме функции (`calc(..)` в следующем листинге):

```
function calculator() {
    // ..
}

var calc = calculator();
```

При каждом вызове `calc(..)` передается один символ, представляющий нажатие кнопки на калькуляторе. Чтобы не усложнять задачу, ограничим наш калькулятор вводом только цифр (0-9), арифметических операций (`+`, `-`, `*`, `/`) и знака `=` для вычисления результата. Операции обрабатываются строго в порядке ввода; не поддерживаются ни группировка `()`, ни приоритеты операторов.

Ввод дробных чисел не поддерживается, но они могут появляться в результате выполнения операции деления. Ввод отрицательных чисел не поддерживается, но они могут появиться в результате выполнения операции вычитания. Таким образом, любое отрицательное или вещественное число можно сгенерировать, выполнив сначала операцию для его вычисления. После этого можно продолжить вычисления с этим значением.

Значение, возвращаемое при вызовах `calc(..)`, должно моделировать показания на дисплее реального калькулятора — либо воспроизводить только что нажатую клавишу, либо вычислять результат при нажатии `=`.

Пример:

```
calc("4"); // 4
calc("+"); // +
calc("7"); // 7
calc("3"); // 3
calc("-"); // -
calc("2"); // 2
calc("="); // 75
calc("*"); // *
calc("4"); // 4
calc("="); // 300
calc("5"); // 5
calc("-"); // -
calc("5"); // 5
calc("="); // 0
```

Так как пользоваться таким калькулятором неудобно, определяется вспомогательная функция `useCalc(..)`, которая последовательно передает калькулятору символы из строки и каждый раз вычисляет выводимое значение:

```
function useCalc(calc,keys) {
    return [...keys].reduce(
        function showDisplay(display,key){
            var ret = String( calc(key) );
            return (
                display +
                (
```

```
                    (ret != "" && key == "=") ?
                        "=" :
                        ""
                ) +
                ret
            );
        },
        ""
    );
}

useCalc(calc,"4+3=");   // 4+3=7
useCalc(calc,"+9=");    // +9=16
useCalc(calc,"*8=");    // *5=128
useCalc(calc,"7*2*3="); // 7*2*3=42
useCalc(calc,"1/0=");   // 1/0=ERR
useCalc(calc,"+3=");    // +3=ERR
useCalc(calc,"51=");    // 51
```

В самом разумном сценарии использования `useCalc(..)` последним вводимым символом всегда должен быть знак `=`.

Форматирование выводимых результатов иногда требует специальной обработки. Я представлю функцию `formatTotal(..)`, которая должна использоваться вашим калькулятором каждый раз, когда он должен вывести текущий вычисленный результат (после ввода `=`):

```
function formatTotal(display) {
    if (Number.isFinite(display)) {
        // ограничить вывод 11 символами
        let maxDigits = 11;
        // зарезервировать место для обозначения "e+"?
        if (Math.abs(display) > 99999999999) {
            maxDigits -= 6;
        }
        // зарезервировать место для "-"?
        if (display < 0) {
            maxDigits--;
        }

        // целое число?
        if (Number.isInteger(display)) {
```

```
            display = display
                .toPrecision(maxDigits)
                .replace(/\.0+$/,"");
        }

        // дробное число
        else {
            // зарезервировать место для "."
            maxDigits--;
            // зарезервировать место для начального "0"?
            if (
                Math.abs(display) >= 0 &&
                Math.abs(display) < 1
            ) {
                maxDigits--;
            }
            display = display
                .toPrecision(maxDigits)
                .replace(/0+$/,"");
        }
    }
    else {
        display = "ERR";
    }
    return display;
}
```

Не отвлекайтесь на то, как работает `formatTotal(..)`. Большая часть ее логики сводится к ограничению вывода калькулятора 11 символами (даже для отрицательных чисел), повторяющихся точек и даже экспоненциальной записи «e+», если она требуется.

И постарайтесь не увязнуть в болоте вокруг специфического поведения калькулятора. Сосредоточьтесь на памяти замыкания.

Попробуйте выполнить упражнение самостоятельно, а затем сверьтесь с предлагаемым решением в конце приложения.

## Модули

В этом упражнении калькулятор из упражнения «Замыкания» (часть 3) преобразуется в модуль.

Мы не будем добавлять в калькулятор никакую дополнительную функциональность, только изменим интерфейс. Вместо вызова одиночной функции `calc(..)` мы будем вызывать конкретные методы открытого API для каждого нажатия клавиши на калькуляторе. Результаты остаются теми же.

Этот модуль должен быть выражен в виде фабричной функции классического модуля с именем `calculator()` вместо одиночного экземпляра IIFE, чтобы при желании можно было создать несколько экземпляров калькулятора.

Открытый API должен включать следующие методы:

- `number(..)` (ввод: нажатый символ/цифра)
- `plus()`
- `minus()`
- `mult()`
- `div()`
- `eq()`

Сценарий использования выглядит примерно так:

```
var calc = calculator();
calc.number("4"); // 4
calc.plus();      // +
calc.number("7"); // 7
calc.number("3"); // 3
calc.minus();     // -
calc.number("2"); // 2
calc.eq();        // 75
```

Функция `formatTotal(..)` остается такой же, как в предыдущем упражнении. Однако вспомогательную функцию `useCalc(..)` необходимо отрегулировать для работы с API модуля:

```
function useCalc(calc,keys) {
    var keyMappings = {
        "+": "plus",
        "-": "minus",
        "*": "mult",
```

```
        "/": "div",
        "=": "eq"
    };

    return [...keys].reduce(
        function showDisplay(display,key){
            var fn = keyMappings[key] || "number";
            var ret = String( calc[fn](key) );
            return (
                display +
                (
                    (ret != "" && key == "=") ?
                        "=" :
                        ""
                ) +
                ret
            );
        },
        ""
    );
}

useCalc(calc,"4+3=");   // 4+3=7
useCalc(calc,"+9=");    // +9=16
useCalc(calc,"*8=");    // *5=128
useCalc(calc,"7*2*3="); // 7*2*3=42
useCalc(calc,"1/0=");   // 1/0=ERR
useCalc(calc,"+3=");    // +3=ERR
useCalc(calc,"51=");    // 51
```

Попробуйте выполнить упражнение самостоятельно, а затем сверьтесь с предлагаемым решением в конце приложения.

Работая над этим упражнением, также выделите немного времени на анализ достоинств и недостатков представления калькулятора в формате модуля в отличие от подхода с функцией/замыканием из предыдущего упражнения.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ: объясните свой ход мыслей в нескольких предложениях.

ДОПОЛНИТЕЛЬНОЕ ЗАДАНИЕ 2: попробуйте преобразовать свой модуль к другому формату модулей, включая UMD, CommonJS и ESM (ES Modules).

# Предлагаемые решения

Надеюсь, вы попробовали самостоятельно выполнить упражнения, прежде чем переходить к ответам. Не мухлевать!

Помните, что каждое предлагаемое решение — всего лишь один из многих возможных вариантов решения задач. Ответы не являются единственно правильными, а всего лишь демонстрируют разумный подход к решению.

Главное, что вы можете сделать с предлагаемыми решениями, — сравнить их со своим кодом и проанализировать, почему мы приняли похожие или разные решения. Не увлекайтесь малозначительными подробностями; постарайтесь сосредоточиться на главной теме.

## Предлагаемое решение: «Камешки и банки»

Решение упражнения «Камешки и банки» может выглядеть так:

```
// КРАСНЫЙ(1)
const howMany = 100;

// Решето Эратосфера
function findPrimes(howMany) {
    // СИНИЙЙ(2)
    var sieve = Array(howMany).fill(true);
    var max = Math.sqrt(howMany);

    for (let i = 2; i < max; i++) {
        // ЗЕЛЕНЫЙ(3)
        if (sieve[i]) {
            // ОРАНЖЕВЫЙ(4)
            let j = Math.pow(i,2);
            for (let k = j; k < howMany; k += i) {
                // PURPLE(5)
                sieve[k] = false;
            }
        }
    }
```

```
    return sieve
        .map(function getPrime(flag,prime){
            // РОЗОВЫЙ  (6)
            if (flag) return prime;
            return flag;
        })
        .filter(function onlyPrimes(v){
            // ЖЕЛТЫЙ(7)
            return !!v;
        })
        .slice(1);
}

findPrimes(howMany);
// [
// 2, 3, 5, 7, 11, 13, 17,
// 19, 23, 29, 31, 37, 41,
// 43, 47, 53, 59, 61, 67,
// 71, 73, 79, 83, 89, 97
// ]
```

## Предлагаемое решение: «Замыкания» (часть 1)

Решение упражнения «Замыкания» (часть 1) для функций `isPrime(..)` и `factorize(..)` может выглядеть так:

```
var isPrime = (function isPrime(v){
    var primes = {};

    return function isPrime(v) {
        if (v in primes) {
            return primes[v];
        }
        if (v <= 3) {
            return (primes[v] = v > 1);
        }
        if (v % 2 == 0 || v % 3 == 0) {
            return (primes[v] = false);
        }
        let vSqrt = Math.sqrt(v);
        for (let i = 5; i <= vSqrt; i += 6) {
            if (v % i == 0 || v % (i + 2) == 0) {
                return (primes[v] = false);
```

```
            }
        }
        return (primes[v] = true);
    };
})();

var factorize = (function factorize(v){
    var factors = {};

    return function findFactors(v) {
        if (v in factors) {
            return factors[v];
        }
        if (!isPrime(v)) {
            let i = Math.floor(Math.sqrt(v));
            while (v % i != 0) {
                i--;
            }
            return (factors[v] = [
                ...findFactors(i),
                ...findFactors(v / i)
            ]);
        }
        return (factors[v] = [v]);
    };
})();
```

Общая последовательность действий, которые я использовал для каждой функции:

1. Упаковать IIFE для определения области видимости, в которой будет находиться переменная с кэшем.
2. В используемом вызове сначала проверить кэш, и если результат уже известен — вернуть его.
3. В каждом месте, где в исходной версии располагалось ключевое слово `return`, присвоить значение элементу кэша и просто вернуть результаты операции присваивания — это трюк для экономии памяти, который в книге используется прежде всего для краткости.

Я также переименовал внутреннюю функцию `factorize(..)` в `findFactors(..)`. Технически это не является необходимым, но помо-

гает более четко выразить, какую функцию вызывают рекурсивные вызовы.

## Предлагаемое решение: «Замыкания» (часть 2)

Решение упражнения «Замыкания» (часть 2) для функции `toggle(..)` может выглядеть так:

```
function toggle(...vals) {
    var unset = {};
    var cur = unset;

    return function next(){
        // сохранить предыдущее значение
        // в конце списка
        if (cur != unset) {
            vals.push(cur);
        }
        cur = vals.shift();
        return cur;
    };
}

var hello = toggle("hello");
var onOff = toggle("on","off");
var speed = toggle("slow","medium","fast");

hello();      // "hello"
hello();      // "hello"

onOff();      // "on"
onOff();      // "off"
onOff();      // "on"

speed();      // "slow"
speed();      // "medium"
speed();      // "fast"
speed();      // "slow"
```

## Предлагаемое решение: «Замыкания» (часть 3)

Решение упражнения «Замыкания» (часть 3) для функции `calculator(..)` может выглядеть так:

```
// см. выше:
//
// function useCalc(..) { .. }
// function formatTotal(..) { .. }

function calculator() {
    var currentTotal = 0;
    var currentVal = "";
    var currentOper = "=";

    return pressKey;

    // ********************

    function pressKey(key){
        // цифра?
        if (/\d/.test(key)) {
            currentVal += key;
            return key;
        }
        // оператор?
        else if (/[+*/-]/.test(key)) {
            // серия из нескольких операций?
            if (
                currentOper != "=" &&
                currentVal != ""
            ) {
                // предполагается нажатие '='
                pressKey("=");
            }
            else if (currentVal != "") {
                currentTotal = Number(currentVal);
            }
            currentOper = key;
            currentVal = "";
            return key;
        }
        // клавиша = ?
        else if (
            key == "=" &&
            currentOper != "="
        ) {
            currentTotal = op(
                currentTotal,
```

```
                currentOper,
                Number(currentVal)
            );
            currentOper = "=";
            currentVal = "";
            return formatTotal(currentTotal);
        }
        return "";
    };

    function op(val1,oper,val2) {
        var ops = {
            // ВНИМАНИЕ: стрелочные функции
            // используются только для краткости
            "+": (v1,v2) => v1 + v2,
            "-": (v1,v2) => v1 - v2,
            "*": (v1,v2) => v1 * v2,
            "/": (v1,v2) => v1 / v2
        };
        return ops[oper](val1,val2);
    }
}

var calc = calculator();

useCalc(calc,"4+3=");        // 4+3=7
useCalc(calc,"+9=");         // +9=16
useCalc(calc,"*8=");         // *5=128
useCalc(calc,"7*2*3=");      // 7*2*3=42
useCalc(calc,"1/0=");        // 1/0=ERR
useCalc(calc,"+3=");         // +3=ERR
useCalc(calc,"51=");         // 51
```

Помните: это упражнение посвящено замыканиям. Основное внимание в нем должно уделяться не фактической механике работы калькулятора, а правильному сохранению состояния калькулятора между вызовами.

## Предлагаемое решение: «Модули»

Решение упражнения «Модули» для функции `calculator(..)` может выглядеть так:

```
// см. выше:
//
// function useCalc(..) { .. }
// function formatTotal(..) { .. }

function calculator() {
    var currentTotal = 0;
    var currentVal = "";
    var currentOper = "=";

    var publicAPI = {
        number,
        eq,
        plus() { return operator("+"); },
        minus() { return operator("-"); },
        mult() { return operator("*"); },
        div() { return operator("/"); }
    };

    return publicAPI;

    // ********************

    function number(key) {
        // цифра?
        if (/\d/.test(key)) {
            currentVal += key;
            return key;
        }
    }

    function eq() {
        // клавиша = ?
        if (currentOper != "=") {
            currentTotal = op(
                currentTotal,
                currentOper,
                Number(currentVal)
            );
            currentOper = "=";
            currentVal = "";
            return formatTotal(currentTotal);
        }
        return "";
    }
```

```
    function operator(key) {
        // серия из нескольких операций?
        if (
            currentOper != "=" &&
            currentVal != ""
        ) {
            // предполагается нажатие '='
            eq();
        }
        else if (currentVal != "") {
            currentTotal = Number(currentVal);
        }
        currentOper = key;
        currentVal = "";
        return key;
    }

    function op(val1,oper,val2) {
        var ops = {
            // ВНИМАНИЕ: стрелочные функции
            // используются только для краткости
            "+": (v1,v2) => v1 + v2,
            "-": (v1,v2) => v1 - v2,
            "*": (v1,v2) => v1 * v2,
            "/": (v1,v2) => v1 / v2
        };
        return ops[oper](val1,val2);
    }
}

var calc = calculator();

useCalc(calc,"4+3=");        // 4+3=7
useCalc(calc,"+9=");         // +9=16
useCalc(calc,"*8=");         // *5=128
useCalc(calc,"7*2*3=");      // 7*2*3=42
useCalc(calc,"1/0=");        // 1/0=ERR
useCalc(calc,"+3=");         // +3=ERR
useCalc(calc,"51=");         // 51
```

И на этом книга завершена — поздравляю вас с достижением!

Увидимся снова!