O'REILLY®

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

— Шейн Хадсон, разработчик веб-сайтов

КАЙЛ СИМПСОН

ЗАМЫКАНИЯ & ОБЪЕКТЫ

ВЫ НЕ ЗНАЕТЕ JS

this & Object Prototypes
Scope & Closures

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo O'REILLY®

ЗАМЫКАНИЯ & ОБЪЕКТЫ

КАЙЛ СИМПСОН

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2019

ББК 32.988-02-018
УДК 004.738.5
С37

Симпсон К.

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

ISBN 978-5-4461-1255-5

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

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

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

ББК 32.988-02-018
УДК 004.738.5

ISBN 978-1491904152 англ.

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

ISBN 978-5-4461-1255-5

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

Оглавление

Введение

С самых первых дней существования Всемирной паутины язык JavaScript стал фундаментальной технологией для управления интерактивностью контента, потребляемого пользователями. Хотя история JavaScript начиналась с мерцающих следов от указателя мыши и раздражающих всплывающих подсказок, через два десятилетия технология и возможности JavaScript выросли на несколько порядков, и лишь немногие сомневаются в его важности как ядра самой распространенной программной платформы в мире: веб-технологий.

Но как язык JavaScript постоянно подвергался серьезной критике — отчасти из-за своего происхождения, еще больше из-за своей философии проектирования. Даже само его название наводит на мысли, как однажды выразился Брендан Эйх (Brendan Eich), о «недоразвитом младшем брате», который стоит рядом со своим старшим и умным братом Java. Тем не менее такое название возникло исключительно по соображениям политики и маркетинга. Между этими двумя языками существуют колоссальные различия. У JavaScript с Java общего не больше, чем у луна-парка с Луной.

Так как JavaScript заимствует концепции и синтаксические идиомы из нескольких языков, включая процедурные корни в стиле C и менее очевидные функциональные корни в стиле Scheme/Lisp, он в высшей степени доступен для широкого спектра разработчиков — даже обладающих минимальным опытом программирования. Программа «Hello World» на JavaScript настолько проста, что

язык располагает к себе и кажется удобным с самых первых минут знакомства.

Пожалуй, JavaScript — один из самых простых языков для изучения и начала работы, но из-за его странностей хорошее знание этого языка встречается намного реже, чем во многих других языках. Если для написания полноценной программы на C или C++ требуется достаточно глубокое знание языка, полномасштабное коммерческое программирование на JavaScript порой (и достаточно часто) едва затрагивает то, на что способен этот язык.

Хитроумные концепции, глубоко интегрированные в язык, проявляются в простых на первый взгляд аспектах, например, передаче функций в форме обратных вызовов. У разработчика JavaScript появляется соблазн просто использовать язык «как есть» и не беспокоиться о том, что происходит «внутри».

Это одновременно простой и удобный язык, находящий повсеместное применение, и сложный, многогранный набор языковых механик, истинный смысл которых без тщательного изучения останется непонятным даже для самого опытного разработчика JavaScript.

В этом заключается парадокс JavaScript; ахиллесова пята языка; проблема, которой мы сейчас займемся. Так как JavaScript можно использовать без полноценного понимания, очень часто понимание языка так и не приходит к разработчику.

Задача

Если каждый раз, сталкиваясь с каким-то сюрпризом или неприятностью в JavaScript, вы заносите их в «черный список» (как многие привыкли делать), вскоре от всей мощи JavaScript у вас останется пустая скорлупа. Хотя это подмножество принято на-

зывать «Хорошими Частями», я призываю вас, дорогой читатель, рассматривать его как «Простые Части», «Безопасные Части» и даже «Неполные Части».

Серия «Вы не знаете JS» идет в прямо противоположном направлении: изучить и глубоко понять весь язык JavaScript, и особенно «Сложные Части».

Здесь мы прямо говорим о существующей среди разработчиков JS тенденции изучать «ровно столько, сколько нужно» для работы, не заставляя себя разбираться в том, что именно происходит и почему язык работает именно так. Более того, мы воздерживаемся от распространенной тактики отступить, когда двигаться дальше становится слишком трудно.

Я не привык останавливаться в тот момент, когда что-то просто работает, а я толком сам не знаю почему, — и вам не советую. Приглашаю вас на путешествие по этим неровным, непростым дорогам; здесь вы узнаете, что собой представляет язык JavaScript и что он может сделать. С этими знаниями ни один метод, ни один фреймворк, ни одно модное сокращение или термин недели не будут за пределами вашего понимания.

В каждой из книг серии мы возьмем одну из конкретных базовых частей языка, которые часто понимаются неправильно или недостаточно глубоко, и рассмотрим ее очень глубоко и подробно. После чтения у вас должна сформироваться твердая уверенность в том, что вы понимаете не только теорию, но и практические аспекты «того, что нужно знать для работы».

Скорее всего, то, что вы сейчас знаете о JavaScript, вы узнавали по частям от других людей, которые тоже недостаточно хорошо разбирались в теме. Такой JavaScript — не более чем тень настоящего языка. На самом деле вы пока JavaScript не знаете, но будете знать, если как следует ознакомитесь с этой серией книг. Читайте дальше, друзья мои. JavaScript ждет вас.

Благодарности

Я должен поблагодарить многих людей, без которых эта книга и вся серия не появились бы на свет.

Прежде всего, я должен поблагодарить свою жену Кристен Симпсон (Christen Simpson) и своих детей Итана и Эмили — они мирились с тем, что папа подолгу сидит за компьютером. Даже когда я не пишу книги, мое увлечение JavaScript приковывает меня к экрану намного чаще, чем следовало бы. Впрочем, именно благодаря времени, позаимствованному у моей семьи, эти книги так глубоко и полно объясняют JavaScript вам, читателю. Я обязан моей семье всем.

Хочу поблагодарить своих редакторов из O'Reilly, а именно Симона Сен-Лорана (Simon St.Laurent) и Брайана Макдональда (Brian MacDonald), а также весь остальной коллектив издательства и специалистов по маркетингу. Работать с ними было невероятно интересно, и они особенно доброжелательно отнеслись к эксперименту с написанием, редактированием и печатью этой книги как проекта «с открытым кодом».

Спасибо всем, кто внес свой вклад в работу над этой серией книг, кто поделился своими предложениями и улучшениями: Шелли Пауэрс (Shelley Powers), Тим Ферро (Tim Ferro), Эван Борден (Evan Borden), Форрест Л. Норвелл (Forrest L Norvell), Дженнифер Дэвис (Jennifer Davis), Джесс Харлин (Jesse Harlin) и многие другие. Спасибо Шейну Хадсону (Shane Hudson) за отличное введение.

Спасибо бесчисленным участникам сообщества, включая представителей комитета TC39. Они делились с нами своими знаниями, терпеливо и подробно отвечали на мои бесчисленные вопросы: Джон-Дэвид Далтон (John-David Dalton), Юрий «kangax» Зайцев (Juriy Zaytsev), Матиас Байненс (Mathias Bynens), Рик

Уолдрон (Rick Waldron), Аксель Раушмайер (Axel Rauschmayer), Николас Закас (Nicholas Zakas), Энгус Кролл (Angus Croll), Джордан Харбанд (Jordan Harband), Дэйв Херман (Dave Herman), Брендан Эйх (Brendan Eich), Аллен Вирфс-Брок (Allen Wirfs-Brock), Брэдли Мек (Bradley Meck), Доменик Деникола (Domenic Denicola), Дэвид Уолш (David Walsh), Тим Дисней (Tim Disney), Крис Ковал (Kris Kowal), Петер ван дер Зее (Peter van der Zee), Андреа Джиаммарчи (Andrea Giammarchi), Кит Кэмбридж (Kit Cambridge)... и так много других, что я не затронул даже вершину айсберга.

Так как серия книг «Вы не знаете JS» родилась на Kickstarter, я также хочу поблагодарить всех моих (почти) 500 щедрых бэкеров, без которых эта серия не могла состояться. Вот они: Ян Шпила (Jan Szpila), nokiko, Мурали Кришнамурти (Murali Krishnamoorthy), Райан Джой (Ryan Joy), Крейг Пэтчетт (Craig Patchett), pdqtrader, Дэйл Фуками (Dale Fukami), Рэй Хэтфилд (ray hatfield), Родриго Перес (R0drigo Perez) [Mx], Дэн Петитт (Dan Petitt), Джек Фрэнклин (Jack Franklin), Эндрю Берри (Andrew Berry), Брайан Гринстед (Brian Grinstead), Роб Сазерленд (Rob Sutherland), Сержи Месегер (Sergi Meseguer), Филип Гурли (Phillip Gourley), Марк Уотсон (Mark Watson), Джефф Карут (Jeff Carouth), Альфредо Сумаран (Alfredo Sumaran), Мартин Сакс (Martin Sachse), Марсио Барриос (Marcio Barrios), Дэн (Dan), AimelyneM, Мэтт Салливан (Matt Sullivan), Делнатт Пьер-Антуан (Delnatte Pierre-Antoine), Джейк Смит (Jake Smith), Юджин Тудорансеа (Eugen Tudorancea), Айрис (Iris), Дэвид Трин (David Trinh), simonstl, Рэй Дэли (Ray Daly), Урос Грубер (Uros Gruber), Джастин Майерс (Justin Myers), Шай Зонис (Shai Zonis), Mom & Dad, Дэвин Кларк (Devin Clark), Дэннис Палмер (Dennis Palmer), Брайан Панахи Джонсон (Brian Panahi Johnson), Джош Маршалл (Josh Marshall), Маршалл (Marshall), Дэннис Керр (Dennis Kerr), Мэтт Стил (Matt Steele), Эрик Слагтер (Erik Slagter), Sacah, Джастин Рэйнбоу (Justin Rainbow), Кристиан Нилссон (Christian

Nilsson), Делапуи (Delapouite), Д. Перейра (D. Pereira), Николас Хойзи (Nicolas Hoizey), Джордж В. Рейли (George V. Reilly), Дэн Ривс (Dan Reeves), Бруно Латернер (Bruno Laturner), Чед Дженнингс (Chad Jennings), Шейн Кинг (Shane King), Джереми Ли Кохик (Jeremiah Lee Cohick), od3n, Стэн Ямейн (Stan Yamane), Марко Вучинич (Marko Vucinic), Jim B, Стивен Коллинз (Stephen Collins), Эгир Торстейнссон (Ægir Þorsteinsson), Эрик Педерсон (Eric Pederson), Овейн (Owain), Нейтан Смит (Nathan Smith), Jeanetteurphy, Александр (Alexandre) ELISÉ, Крис Петерсон (Chris Peterson), Рик Уотсон (Rik Watson), Люк Мэтьюз (Luke Matthews), Джастин Лоуэри (Justin Lowery), Мортен Нильсен (Morten Nielsen), Вернон Кеснер (Vernon Kesner), Четан Шеной (Chetan Shenoy), Пол Трегоинг (Paul Tregoing), Марк Грабански (Marc Grabanski), Дион Альмейр (Dion Almaer), Эндрю Салливан (Andrew Sullivan), Кейт Элзасс (Keith Elsass), Том Берк (Tom Burke), Брайан Эшенфелтер (Brian Ashenfelter), Дэвид Стюарт (David Stuart), Карл Сведберг (Karl Swedberg), Грэм (Graeme), Брэндон Хейс (Brandon Hays), Джон Кристофер (John Christopher), Gior, manoj reddy, Чед Смит (Chad Smith), Джаред Харбор (Jared Harbour), Минору Тода (Minoru TODA), Крис Уигли (Chris Wigley), Дэниел Ми (Daniel Mee), Майк (Mike), Handyface, Алекс Яраус (Alex Jahraus), Карл Фурроу (Carl Furrow), Роб Фулкрод (Rob Foulkrod), Макс Шишкин (Max Shishkin), Ли Пенни мл. (Leigh Penny Jr.), Роберт Фергусон (Robert Ferguson), Майк ван Хенселаар (Mike van Hoenselaar), Хасс Шугаард (Hasse Schougaard), Раджан Венкатагуру (rajan venkataguru), Джефф Адамс (Jeff Adams), Трей Роббинс (Trae Robbins), Рольф Лангенхузен (Rolf Langenhuijzen), Хорхе Антунес (Jorge Antunes), Алекс Колосков (Alex Koloskov), Хью Гриниш (Hugh Greenish), Тим Джонс (Tim Jones), Хосе Очоа (Jose Ochoa), Майкл Бреннан-Уайт (Michael Brennan-White), Нага Хариш Муввa (Naga Harish Muvva), Баркоци Давид (Barkóczi Dávid), Китт Ходсден (Kitt Hodsden), Пол Макгроу (Paul McGraw), Саша Голдхофер (Sascha Goldhofer), Эндрю Меткаф (Andrew Metcalf), Маркус Крог

(Markus Krogh), Майкл Мэтьюз (Michael Mathews), Мэтт Джаред (Matt Jared), Juanfran, Джорджи Киршнер (Georgie Kirschner), Кенни Ли (Kenny Lee), Тед Чжан (Ted Zhang), Амит Пахва (Amit Pahwa), Инбал Синаи (Inbal Sinai), Дэн Рейн (Dan Raine), Шабсе Лакс (Schabse Laks), Майкл Терворт (Michael Tervoort), Александр Абро (Alexandre Abreu), Алан Джозеф Уильямс (Alan Joseph Williams), NicolasD, Синди Вонг (Cindy Wong), Рег Брайтуэйт (Reg Braithwaite), LocalPCGuy, Джон Фрискис (Jon Friskics), Крис Мерриман (Chris Merriman), Джон Пена (John Pena), Джейкоб Кац (Jacob Katz), Сью Локвуд (Sue Lockwood), Магнус Йоханссон (Magnus Johansson), Джереми Крэпси (Jeremy Crapsey), Гжегож Павловски (Grzegorz Pawłowski), Нико Нуззачи (nico nuzzaci), Кристин Уилкс (Christine Wilks), Ханс Бергрен (Hans Bergren), Чарльз Монтгомери (charles montgomery), Ариэль Фогел (Ariel דב-בבל Fogel), Иван Колев (Ivan Kolev), Дэниел Кампос (Daniel Campos), Хью Вуд (Hugh Wood), Кристиан Брэдфорд (Christian Bradford), Фредерик Харпер (Frédéric Harper), Ионут Дан Попа (Ionuț Dan Popa), Джефф Тримбл (Jeff Trimble), Руперт Вуд (Rupert Wood), Трей Каррико (Trey Carrico), Панчо Лопес (Pancho Lopez), Джоэл Куйтен (Joël kuijten), Том А. Марра (Tom A Marra), Джефф Джуисс (Jeff Jewiss), Джейкоб Риос (Jacob Rios), Паоло Ди Стефано (Paolo Di Stefano), Соледад Пенадес (Soledad Penades), Крис Гербер (Chris Gerber), Андрей Долганов (Andrey Dolganov), Уилл Мур III (Wil Moore III), Томас Мартино (Thomas Martineau), Карим (Kareem), Бен Type (Ben Thouret), Уди Нир (Udi Nir), Морган Лаупис (Morgan Laupies), Джори Карсон-Берсон (jory carson-burson), Натан Л. Смит (Nathan L. Smith), Эрик Дэймон Уолтерс (Eric Damon Walters), Дерри Лозано-Хойленд (Derry Lozano-Hoyland), Джеоффри Уайзман (Geoffrey Wiseman), mkeehner, KatieK, Скотт Макфарлейн (Scott MacFarlane), Брайан Лашомб (Brian LaShomb), Адриен Мас (Adrien Mas), Кристофер Росс (christopher ross), Иэн Литтман (Ian Littman), Дэн Аткинсон (Dan Atkinson), Эллиот Джоуб (Elliot Jobe), Ник Дозье (Nick Dozier), Питер Вули (Peter Wooley), Джон Гувер (John Hoover),

dan, Мартин Э. Джексон (Martin A. Jackson), Гектор Фернандо Хуртадо (Héctor Fernando Hurtado), Энди Эннаморато (andy ennamorato), Пол Селтман (Paul Seltmann), Мелисса Гор (Melissa Gore), Дэйв Поллард (Dave Pollard), Джек Смит (Jack Smith), Филип Да Силва (Philip Da Silva), Гай Израэли (Guy Israeli), @megalithic, Дэмиан Кроуфорд (Damian Crawford), Феликс Глише (Felix Gliesche), Эйприл Картер Грант (April Carter Grant), Хайди (Heidi), Джим Тирни (jim tierney), Андреа Джиаммарчи (Andrea Giammarchi), Нико Виньола (Nico Vignola), Дон Джонс (Don Jones), Крис Хартьес (Chris Hartjes), Алекс Хоуз (Alex Howes), Джон Гиббон (john gibbon), Дэвид Дж. Грум (David J. Groom), BBox, Ю Дилис Сун (Yu Dilys Sun), Нэйт Стейнер (Nate Steiner), Брэндон Сатром (Brandon Satrom), Брайан Уайант (Brian Wyant), Уэсли Хейлз (Wesley Hales), Иэн Паунси (Ian Pouncey), Тимоти Кевин Оксли (Timothy Kevin Oxley), Джордж Терезакис (George Terezakis), Санджай Радж (sanjay raj), Джордан Харбанд (Jordan Harband), Марко Маклайон (Marko McLion), Вольфганг Кауфман (Wolfgang Kaufmann), Паскаль Пеккерт (Pascal Peuckert), Дэйв Нагент (Dave Nugent), Маркус Либелт (Markus Liebelt), Уэллинг Гусман (Welling Guzman), Ник Кули (Nick Cooley), Дэниел Мескита (Daniel Mesquita), Роберт Сайварт (Robert Syvarth), Крис Койе (Chris Coyier), Реми Бах (Rémy Bach), Адам Дугал (Adam Dougal), Алистер Даггин (Alistair Duggin), Дэвид Лойдолт (David Loidolt), Эд Ричер (Ed Richer), Брайан Шено (Brian Chenault), GoldFire Studios, Карлес Андре (Carles Andrés), Карлос Кабо (Carlos Cabo), Юя Сайто (Yuya Saito), Роберто Рикардо (roberto ricardo), Барнетт Клейн (Barnett Klane), Майк Мур (Mike Moore), Кевин Маркс (Kevin Marx), Джастин Лав (Justin Love), Джо Тейлор (Joe Taylor), Пол Дижу (Paul Dijou), Майкл Колер (Michael Kohler), Роб Кэсси (Rob Cassie), Майк Тирни (Mike Tierney), Коди Лерой Линдли (Cody Leroy Lindley), tofuji, Шимон Шварц (Shimon Schwartz), Рэймонд (Raymond), Люк Де Бруве (Luc De Brouwer), Дэвид Хейс (David Hayes), Рис Бретт-Боуэн (Rhys Brett-Bowen), Дмитрий (Dmitry), Азиз Хури (Aziz Khoury), Дин (Dean), Скотт

Толински (Scott Tolinski Level Up), Клеман Буари (Clement Boirie), Джордже Лукич (Djordje Lukic), Антон Котенко (Anton Kotenko), Рафаэль Коррал (Rafael Corral), Филипп Гурвиц (Philip Hurwitz), Джонатан Пиджин (Jonathan Pidgeon), Джейсон Кэмпбелл (Jason Campbell), Джозеф С. (Joseph C.), SwiftOne, Иэн Хонер (Jan Hohner), Дерик Бэйли (Derick Bailey), getify, Дэниел Кузино (Daniel Cousineau), Крис Чарлтон (Chris Charlton), Эрик Тернер (Eric Turner), Дэвид Тернер (David Turner), Джоэл Галеран (Joël Galeran), Dharma Vagabond, Адам (adam), Дирк ван Берген (Dirk van Bergen), dave ♥♫★ furf, Ведран Закани (Vedran Zakanj), Райан Макаллен (Ryan McAllen), Натали Пэтрис Такер (Natalie Patrice Tucker), Эрик Дж. Бивона (Eric J. Bivona), Адам Спунер (Adam Spooner), Аарон Кавано (Aaron Cavano), Келли Пэкер (Kelly Packer), Эрик Джней (Eric J), Мартин Дреновак (Martin Drenovac), Эмилис (Emilis), Майкл Пеликан (Michael Pelikan), Скотт Ф. Уолтер (Scott F. Walter), Джош Фримен (Josh Freeman), Брэндон Хадженс (Brandon Hudgeons), Виджай Ченнупати (vijay chennupati), Билл Гленнон (Bill Glennon), Робин Р. (Robin R.), Трой Форстер (Troy Forster), otaku_coder, Брэд (Brad), Скотт (Scott), Фредерик Острандер (Frederick Ostrander), Адам Брилл (Adam Brill), Себ Флиппенс (Seb Flippence), Майкл Андерсон (Michael Anderson), Джейкоб (Jacob), Адам Рэндлетт (Adam Randlett), Standard, Джошуа Клэнтон (Joshua Clanton), Себастиан Куба (Sebastian Kouba), Крис Дек (Chris Deck), SwordFire, Ханнес Папенберг (Hannes Papenberg), Ричард Вобер (Richard Woeber), hnzz, Роб Кроутер (Rob Crowther), Джедидайя Бродбент (Jedidiah Broadbent), Сергей Чернышев (Sergey Chernyshev), Джей-Эр Хамон (Jay-Ar Jamon), Бен Комби (Ben Combee), Лучиано Боначела (luciano bonachela), Марк Томлинсон (Mark Tomlinson), Кит Кэмбридж (Kit Cambridge), Майкл Мелгарес (Michael Melgares), Джейкоб Адамс (Jacob Adams), Адриан Бруинхаут (Adrian Bruinhout), Бев Вибер (Bev Wieber), Скотт Пулео (Scott Puleo), Томас Херцог (Thomas Herzog), Эйприл Леоне (April Leone), Дэниел Мизилински (Daniel Mizieliński), Кеес ван Гинкел

(Kees van Ginkel), Джон Абрамс (Jon Abrams), Эрвин Хайзер (Erwin Heiser), Ави Лавиад (Avi Laviad), Дэвид Ньюэлл (David Newell), Жан-Франсуа Тюрко (Jean-Francois Turcot), Нико Робертс (Niko Roberts), Эрик Дана (Erik Dana), Чарльз Нилл (Charles Neill), Аарон Холмс (Aaron Holmes), Гжегож Циолковски (Grzegorz Ziółkowski), Нейтан Янгман (Nathan Youngman), Тимоти (Timothy), Джейкоб Мазер (Jacob Mather), Майкл Аллен (Michael Allan), Мохит Сет (Mohit Seth), Райан Эвинг (Ryan Ewing), Бенджамин Ван Тризе (Benjamin Van Treese), Марсело Сантос (Marcelo Santos), Денис Вольф (Denis Wolf), Фил Киз (Phil Keys), Крис Юнг (Chris Yung), Тимо Тийхоф (Timo Tijhof), Мартин Леквалл (Martin Lekvall), Agendine, Грег Уитворт (Greg Whitworth), Хелен Хамфри (Helen Humphrey), Дугал Кэмпбелл (Dougal Campbell), Йоханнес Харт (Johannes Harth), Бруно Гирин (Bruno Girin), Брайан Хоу (Brian Hough), Даррен Ньютон (Darren Newton), Крейг Макфит (Craig McPheat), Оливье Тилл (Olivier Tille), Деннис Ротиг (Dennis Roethig), Матиас Байненс (Mathias Bynens), Брендан Стромбергер (Brendan Stromberger), sundeep, Джон Мейер (John Meyer), Рон Мэйл (Ron Male), Джон Ф. Кростон III (John F Croston III), gigante, Карл Бергенхэм (Carl Bergenhem), Б. Дж. Мэй (B.J. May), Ребека Тайлер (Rebekah Tyler), Тед Фоксберри (Ted Foxberry), Джордан Риз (Jordan Reese), Терри Сьютор (Terry Suitor), afeliz, Том Кифер (Tom Kiefer), Даррах Даффи (Darragh Duffy), Кевин Вандербекен (Kevin Vanderbeken), Энди Пирсон (Andy Pearson), Саймон Макдональд (Simon Mac Donald), Абид Дин (Abid Din), Крис Джоэл (Chris Joel), Томас Тониссен (Tomas Theunissen), Дэвид Дик (David Dick), Пол Грок (Paul Grock), Брэндон Вуд (Brandon Wood), Джон Вайс (John Weis), dgrebb, Ник Дженкинс (Nick Jenkins), Чак Лейн (Chuck Lane), Джонни Мегахан (Johnny Megahan), marzsman, Тату Тамминен (Tatu Tamminen), Джеоффри Кнаут (Geoffrey Knauth), Александр Тармолов (Alexander Tarmolov), Джереми Таймс (Jeremy Tymes), Чед Олд (Chad Auld), Шон Пармели (Sean Parmelee), Роб Стенке (Rob Staenke), Дэн Бендер (Dan Bender), Янник Дерва (Yannick

Derwa), Джошуа Джонс (Joshua Jones), Герт Плейзир (Geert Plaisier), Том Лезотт (Tom LeZotte), Кристен Симпсон (Christen Simpson), Стефан Брувик (Stefan Bruvik), Джастин Фальконе (Justin Falcone), Карлос Сантана (Carlos Santana), Майкл Вайс (Michael Weiss), Пабло Виллослада (Pablo Villoslada), Петер де-Хаан (Peter deHaan), Димитрис Илиопулос (Dimitris Iliopoulos), seyDoggy, Адам Джорденс (Adam Jordens), Ной Кантровиц (Noah Kantrowitz), Эмол М. (Amol M), Мэтью Уиннард (Matthew Winnard), Дирк Гинейдер (Dirk Ginader), Финэм Буи (Phinam Bui), Дэвид Рэпсон (David Rapson), Эндрю Бакстер (Andrew Baxter), Флориан Бугел (Florian Bougel), Майкл Джордж (Michael George), Альбан Эскалье (Alban Escalier), Дэниел Селлерс (Daniel Sellers), Саша Рудан (Sasha Rudan), Джон Грин (John Green), Роберт Ковальски (Robert Kowalski), Дэвид И. Тезейра (David I. Teixeira), @ditma, Чарльз Карпентер (Charles Carpenter), Джастин Йост (Justin Yost), Сэм С. (Sam S), Денис Чиккале (Denis Ciccale), Кевин Шорс (Kevin Sheurs), Янник Круассан (Yannick Croissant), По Фрейсес (Pau Fracés), Стивен Макгоуэн (Stephen McGowan), Шон Сирси (Shawn Searcy), Крис Раппел (Chris Ruppel), Кевин Лэмпинг (Kevin Lamping), Джессика Кэмпбелл (Jessica Campbell), Кристофер Шмитт (Christopher Schmitt), Sablons, Джонатан Рейсдорф (Jonathan Reisdorf), Банни Гек (Bunni Gek), Тедди Хафф (Teddy Huff), Майкл Маллани (Michael Mullany), Майкл Фюрстенберг (Michael Fürstenberg), Карл Хендерсон (Carl Henderson), Рик Йостинг (Rick Yoesting), Скотт Николс (Scott Nichols), Эрнан Сьюдад (Hernán Ciudad), Эндрю Майер (Andrew Maier), Майк Стапп (Mike Stapp), Джесси Шоул (Jesse Shawl), Серхио Лопес (Sérgio Lopes), jsulak, Шон Прайс (Shawn Price), Джоэл Клермон (Joel Clermont), Крис Ридманн (Chris Ridmann), Шон Тимм (Sean Timm), Джейсон Финч (Jason Finch), Эйден Монтгомери (Aiden Montgomery), Элайджа Мэнор (Elijah Manor), Дерек Гатрайт (Derek Gathright), Джесси Харлин (Jesse Harlin), Диллон Карри (Dillon Curry), Кортни Майерс (Courtney Myers), Диего Каденас (Diego Cadenas), Арн де Бри (Arne de Bree), Жоао Пауло Дуба (João Paulo Dubas), Джеймс

Тейлор (James Taylor), Филипп Креутли (Philipp Kraeutli), Михай Паун (Mihai Păun), Сэм Гарегозлу (Sam Gharegozlou), joshjs, Мэтт Мерчисон (Matt Murchison), Эрик Уиндэм (Eric Windham), Тимо Берманн (Timo Behrmann), Эндрю Холл (Andrew Hall), Джошуа Прайс (joshua price) и Теофил Виллар (Théophile Villard).

Эта книга писалась на условиях открытого доступа к информации, включая редактирование и печать. Мы многим обязаны репозиторию Github, благодаря которому перед сообществом открылись такие возможности!

Спасибо всем бесчисленным людям, которых я не назвал, но которые заслуживают благодарности. Пусть эта книжная серия «принадлежит» всем нам, и пусть она внесет свой вклад в популяризацию и понимание языка JavaScript к пользе всех нынешних и будущих участников сообщества.

О книге

JavaScript — замечательный язык. Его легко изучать частично и намного сложнее изучать полностью (или хотя бы в достаточной мере). Когда разработчики сталкиваются с трудностями, они обычно винят в этом язык вместо своего ограниченного понимания. В этих книгах я постараюсь исправить такое положение дел и помогу оценить по достоинству язык, который вы можете (и должны) знать достаточно глубоко.

Многие примеры, приведенные в книге, рассчитаны на современные (и обращенные в будущее) среды JavaScript, такие как ES6. При запуске в более старых версиях движка (предшествующих ES6) поведение некоторых примеров может отличаться от описанного в тексте.

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

`Моноширинный шрифт`

Применяется для оформления листингов программ и программных элементов внутри обычного текста, таких как имена переменных и функций, базы данных, типы данных, переменные окружения, инструкции и ключевые слова.

`Моноширинный курсив`

Обозначает текст, который должен замещаться фактическими значениями, вводимыми пользователем или определяемыми из контекста.

Так выделяются советы и предложения.

Так обозначаются советы, предложения и примечания общего характера.

Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

Вспомогательные материалы (примеры кода, упражнения и т. д.) доступны для загрузки по адресу http://bit.ly/1c8HEWF и http://bit.ly/ydkjs-this-code.

Эта книга призвана оказать вам помощь в решении ваших задач. В общем случае все примеры кода из этой книги вы можете использовать в своих программах и в документации. Вам не нужно обращаться в издательство за разрешением, если вы не собираетесь воспроизводить существенные части программного кода. Например, если вы разрабатываете программу и используете в ней несколько отрывков программного кода из книги, вам не нужно обращаться за разрешением. Однако в случае продажи или распространения компакт-дисков с примерами из этой книги вам необходимо получить разрешение от издательства O'Reilly. Если вы отвечаете на вопросы, цитируя данную книгу или примеры из нее, получение разрешения не требуется. Но при включении существенных объемов программного кода примеров из этой книги в вашу документацию вам необходимо будет получить разрешение издательства.

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.

ЧАСТЬ 1

Область видимости и замыкания

Предисловие

В детстве мне нравилось разбирать и собирать всякие устройства: старые мобильные телефоны, стереосистемы и вообще все, до чего я мог добраться. Я был еще слишком мал, чтобы нормально пользоваться гаджетами, но каждый раз, когда что-то ломалось, я сразу же спрашивал у родителей, можно ли мне разобраться, как оно работает.

Помню, однажды я рассматривал печатную плату старого радиоприемника. На ней была какая-то странная длинная трубка, обмотанная медным проводом. Я не мог понять ее назначение и тут же перешел в режим исследования. Что она делает? Зачем она в приемнике? Она не похожа на другие части платы, почему? А для чего ее обмотали медным проводом? Что будет, если снять провод? Теперь я знаю, что это была рамочная антенна, которая строится наматыванием медного провода на ферритовый сердечник и которая часто используется в транзисторных приемниках.

А вам интересно пытаться искать ответы на все «почему»? Большинству детей это интересно. Пожалуй, мне это больше всего нравится в детях — желание узнавать новое.

К сожалению, теперь я считаюсь профессионалом и провожу будни за созданием новых вещей. Когда я был молод, мне нравилась представлять, что когда-нибудь я буду создавать те вещи, которые разбираю сегодня. Конечно, большинство того, что я создаю сегодня, представляет собой код JavaScript, а не ферритовые сердечники... но не так уж далеко код от них ушел! Но хотя мне когда-то нравилась идея создания, сейчас мне часто не хватает

желания разбирать и разбираться. Конечно, я часто выясняю лучший способ решения задачи или исправления ошибок, но редко трачу время на то, чтобы ставить под сомнение свои инструменты.

Именно по этой причине мне так интересна серия книг «Вы не знаете JS». Это *правильно*. Я не знаю JS. Я использую JavaScript в своей повседневной работе, занимаюсь этим уже много лет, но действительно ли понимаю этот язык? Нет. Конечно, я понимаю многие его стороны, часто читаю спецификации и рассылки, но нет — я не понимаю его в той мере, как того желал бы мой внутренний двойник в шестилетнем возрасте.

Книга, которую вы держите в руках, — прекрасное начало серии. Она ориентирована на таких людей, как я (и будем надеяться, и вы). Она не учит вас JavaScript с нуля. Она показывает, насколько мало вы знаете о внутреннем устройстве языка. Кроме того, книга вышла в идеальный момент: стандарт ES6 постепенно приживается, а в разных браузерах успешно идет работа над его реализацией. Если вы еще не взялись за изучение новых возможностей (таких, как `let` и `const`), это издание станет отличным введением в тему.

Итак, я надеюсь, что книга вам понравится. Но я еще сильнее надеюсь, что подход Кайла, его привычка критически осмысливать каждый крошечный фрагмент языка, закрепится у вас в мозгу и в общем рабочем процессе. Не используйте антенну просто так, разберитесь в том, как и почему она работает.

Шейн Хадсон (Shane Hudson)
www.shanehudson.net

1 Что такое область видимости?

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

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

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

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

Но где и как задаются правила области видимости?

Немного теории компиляторов

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Разбираемся в областях видимости

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

Участники

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

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

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

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

Туда и обратно

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

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

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

Вместо этого Компилятор действует так:

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

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

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

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

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

Немного терминологии

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

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

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

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

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

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

А теперь разберемся подробнее.

В следующей команде:

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

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

Сравните:

```
a = 2;
```

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

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

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

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

foo( 2 );
```

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

Здесь тоже выполняется неочевидное, но важное присваивание.

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

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

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

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

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

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

Общение Движка с Областью видимости

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

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

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

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

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

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

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

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

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

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

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

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

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

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

...

Упражнение

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

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

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

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

Вложенная область видимости

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

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

Пример:

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

```
}

var b = 2;

foo( 2 ); // 4
```

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

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

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

Область видимости: Нет, впервые слышу про такое.

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

Область видимости: Ага, знаю. Вот, держи.

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

Метафоры

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

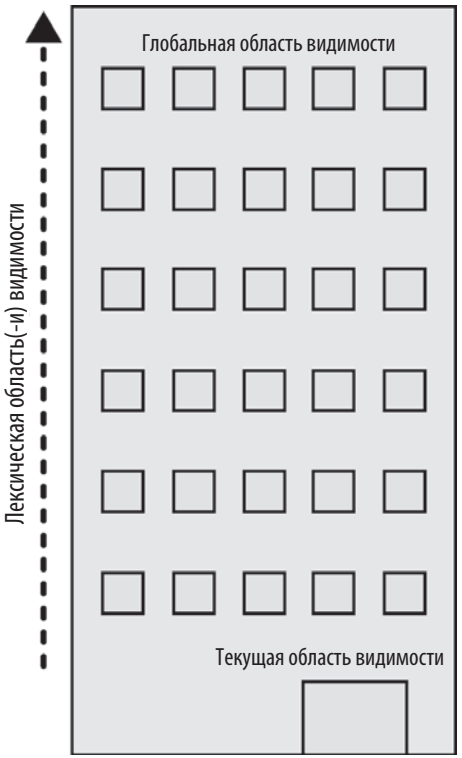

Рис. 1.1

Это здание изображает набор правил вложенных областей видимости. Первый этаж — текущая область видимости, какой бы она ни была, а верхний этаж — глобальная область видимости.

Чтобы разрешить LHS- или RHS-ссылку, система начинает поиск с текущего этажа. Если переменная не будет найдена, поиск поднимается на следующий этаж, ищет там, потом на следующем и т. д. Добравшись до верхнего этажа (глобальной области видимости), вы либо находите искомое, либо не находите. В любом случае здесь придется остановиться.

Ошибки

Почему важно отличать LHS от RHS?

Потому что эти два типа поиска по-разному ведут себя в ситуации, когда переменная еще не была объявлена (не найдена ни в одной из просмотренных областей видимости).

Пример:

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

foo( 2 );
```

Когда RHS-поиск для `b` происходит впервые, он завершается неудачей. Переменная, не найденная в области видимости, считается «необъявленной».

Если RHS-поиск не находит переменную ни в одной из вложенных областей видимости, движок выдает ошибку `ReferenceError`. Важно заметить, что ошибка относится именно к типу `ReferenceError`.

С другой стороны, если движок выполняет LHS-поиск и прибывает на верхний этаж (глобальная область видимости), так и не обнаружив искомое, если программа не выполняется в строгом режиме `strict`[1], в глобальной области видимости создается новая переменная с указанным именем, которая передается движку.

«Нет, такой переменной не было, но я хочу помочь и создать ее для тебя».

[1] См. описание на сайте MSDN (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Strict_mode).

Режим `strict`, добавленный в ES5, во многих отношениях отличается от обычного/нестрогого режима. Одно из отличий заключается в том, что он запрещает автоматическое/неявное создание глобальных переменных. В этом случае LHS-поиск не вернет переменную с глобальной областью видимости, и движок выдаст ошибку `ReferenceError` по аналогии со случаем RHS.

Если переменная будет найдена для RHS-поиска, но вы пытаетесь сделать с ее значением нечто невозможное (например, попытка выполнить как функцию значение, которое функцией не является, или обращение к свойству для значения `null` или `undefined`), движок выдаст другую разновидность ошибки — `TypeError`.

Ошибка `ReferenceError` относится к проблемам при разрешении области видимости, а ошибка `TypeError` подразумевает, что разрешение области видимости прошло успешно, но была сделана попытка выполнить с результатом недопустимую/невозможную операцию.

Итоги

Область видимости — набор правил, определяющих, где и как осуществляется поиск переменной (идентификатора). Поиск может выполняться для цели присваивания переменной (LHS, или левосторонняя ссылка) или же для цели чтения ее значения (RHS, или правосторонняя ссылка).

LHS-ссылки появляются в результате операций присваивания. Присваивания, связанные с областью видимости, могут происходить либо в операторе `=`, либо при передаче аргументов параметрам функции.

Движок JavaScript сначала компилирует код перед выполнением. При этом команды вида `var a = 2;` разбиваются на две части:

1. Сначала `var a` для объявления переменной в области видимости. Этот шаг выполняется перед выполнением кода.

2. Потом `a = 2` для поиска переменной (LHS-ссылка) и присваивания ей, если переменная будет успешно найдена.

Поиск по LHS- и RHS-ссылкам начинается с текущей области видимости. При необходимости (то есть если искомый идентификатор не будет найден) поиск поднимается вверх от вложенной области видимости, по одной области видимости (этажу) за раз, пока не доберется до глобальной области видимости (верхнего этажа). Здесь поиск останавливается: либо идентификатор найден, либо нет.

Для неразрешенных RHS-ссылок выдается исключение `ReferenceError`. Для неразрешенных LHS-ссылок автоматически создается глобальная переменная с заданным именем (если не действует режим `strict`), или происходит ошибка `ReferenceError` (в режиме `strict`).

Ответ на упражнение

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

var c = foo( 2 );
```

1. Найдите все LHS-поиски (всего 3).

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

2. Найдите все RHS-поиски (всего 4).

   `foo(2..`, `= a;`, `a ..` и `.. b`

2 Лексическая область видимости

В главе 1 область видимости была определена как набор правил, управляющих тем, как движок ищет переменную по ее идентификатору и находит ее в текущей области видимости или любой из внешних областей видимости, в которых она содержится.

Существуют две основные модели работы области видимости. Первая модель встречается намного чаще и используется в подавляющем большинстве языков программирования. Она называется *лексической областью видимости* (lexical scope); в этой главе она будет рассмотрена подробно.

Другая модель, которая до сих пор используется в некоторых языках (например, в сценариях Bash, некоторых режимах Perl и т. д.), называется *динамической областью видимости* (dynamic scope). Динамическая область видимости рассматривается в приложении А. Я упоминаю ее здесь только для того, чтобы сравнить ее с лексической областью видимости — моделью, используемой в JavaScript.

Стадия лексического анализа

Как обсуждалось в главе 1, первая традиционная фаза работы стандартного компилятора называется *разбиением на токены* (или *лексическим анализом*). Напомню, что процесс лексического анализа изучает последовательность символов исходного кода и назначает семантический смысл токенам в результате разбора с учетом состояния. Именно эта концепция закладывает фундамент для понимания того, что такое лексическая область видимости и откуда берется это название.

Лексической областью видимости называется область видимости, определяемая на стадии лексического анализа. Другими словами, лексическая область видимости определяется тем, где вы разместили переменные и блоки области видимости во время написания программы, а следовательно, (в основном) жестко фиксируется на момент обработки вашего кода лексическим анализатором.

Вскоре вы увидите, что лексическую область видимости можно «обмануть», то есть изменить ее после обработки кода лексическим анализатором, но делать это не рекомендуется. Старайтесь интерпретировать лексическую область видимости как чисто лексическую, а следовательно, полностью определяемую на стадии написания кода.

Рассмотрим следующий блок кода:

```
function foo(a) {

    var b = a * 2;

    function bar(c) {
        console.log( a, b, c );
    }
```

```
    bar( b * 3 );
}

foo( 2 ); // 2, 4, 12
```

В этом примере существуют три области видимости. Для наглядности можно представить их как пузыри, вложенные друг в друга (рис. 2.1).

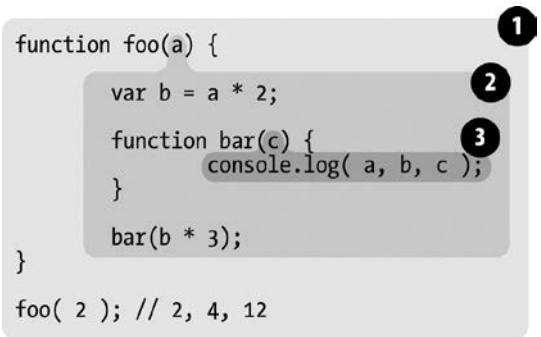

Рис. 2.1

Пузырь 1 охватывает глобальную область видимости и содержит всего один идентификатор: `foo`.

Пузырь 2 охватывает область видимости `foo` и содержит три идентификатора: `a`, `bar` и `b`.

Пузырь 3 охватывает область видимости `bar` и включает один идентификатор: `c`.

Пузыри областей видимости определяются тем, где располагаются блоки, какой из них вложен в другой и т. д. В следующей главе мы обсудим различные единицы области видимости, а пока будем считать, что каждая функция создает новый пузырь области видимости.

Пузырь для `bar` полностью содержится внутри пузыря для `foo`, потому что (и только по этой причине) мы решили определить функцию `bar` именно здесь.

Обратите внимание на строгую вложенность пузырей. Структура не похожа на диаграммы Венна, в которых возможно пересечение пузырей. Другими словами, пузырь некоторой функции не может находиться одновременно в двух других пузырях внешней области действия, и никакая функция не может частично находиться внутри двух других родительских функций.

Поиск

Структура и относительное размещение пузырей областей видимости полностью объясняет движку, где он должен искать идентификатор. В предыдущем фрагменте кода движок выполняет команду `console.log(..)` и переходит к поиску трех переменных, `a`, `b` и `c`. Поиск начинается с внутренней области видимости, то есть области видимости функции `bar(..)`. Здесь найти переменную `a` не удается, поэтому поиск поднимается на один уровень к ближайшей области видимости, то есть области видимости `foo(..)`. Здесь находится переменная `a`, поэтому движок использует эту переменную `a`. То же самое происходит с `b`. Но переменная `c` обнаруживается внутри `bar(..)`.

Если бы переменная `c` присутствовала и внутри `bar(..)`, и внутри `foo(..)`, команда `console.log(..)` нашла и использовала бы переменную из `bar(..)`, так и не добравшись до переменной из `foo(..)`.

Поиск по областям видимости останавливается при нахождении первого совпадения. Одно имя идентификатора может быть задано на нескольких уровнях вложенных областей видимости, что называется «замещением» (внутренний идентификатор «замещает» внешний идентификатор). Независимо от замещения поиск

области видимости всегда начинается с внутренней области видимости, выполняемой в настоящее время, и перемещается наружу/вверх до первого совпадения, где останавливается.

Глобальные переменные также автоматически являются свойствами глобального объекта (`window` в браузерах и т. д.). Это означает, что на глобальную переменную можно ссылаться не только напрямую по ее лексическому имени, но и косвенно через свойство глобального объекта:

```
window.a
```

Этот синтаксис открывает доступ к глобальной переменной, которая в противном случае была бы недоступна из-за замещения. Неглобальные замещенные переменные остаются недоступными.

Неважно, *где* вызывается функция, и даже *как* она вызывается — ее лексическая область видимости определяется только тем, где была объявлена функция.

Лексическая область видимости применяется *только* к полноценным идентификаторам, таким как `a`, `b` и `c`. Если фрагмент кода содержит ссылку `foo.bar.baz`, поиск лексической области видимости будет относиться к идентификатору `foo`, но после того, как переменная будет обнаружена, вступят в силу правила обращения к свойствам объектов для разрешения имен свойств `bar` и `baz` соответственно.

Искажение лексической области видимости

Если лексическая область видимости определяется только местом объявления функции (которое выбирается исключительно во

время написания программы), как возможно модифицировать (то есть фактически исказить) лексическую область видимости во время выполнения?

В JavaScript существуют два таких механизма. Оба в равной степени осуждаются сообществом разработчиков как признаки плохого стиля в вашем коде. Однако типичные аргументы против них часто упускают из виду самое важное обстоятельство: *искажение лексической области видимости ведет к снижению быстродействия*.

Но прежде чем объяснять проблемы с быстродействием, разберемся с тем, как работают эти два механизма.

eval

Функция `eval(..)` в JavaScript получает строковый аргумент и интерпретирует содержимое строки так, словно это реальный код в текущей точке программы. Иначе говоря, вы можете на программном уровне генерировать код внутри своей программы и выполнять сгенерированный код так, словно вы сами его написали.

Если рассматривать `eval(..)` в этом свете, станет ясно, как `eval()` позволяет изменить окружение лексической области видимости. В строках кода, следующих за выполнением `eval(..)`, движок не будет «знать», что код был интерпретирован динамически, а следовательно, изменил окружение лексической области видимости. Движок просто выполнит поиск по лексической области видимости так, как он это делает всегда.

Рассмотрим следующий код:

```
function foo(str, a) {
    eval( str ); // изменение!
    console.log( a, b );
```

```
}
var b = 2;
foo( "var b = 3;", 1 ); // 1, 3
```

Строка `"var b = 3;"` в точке вызова `eval(..)` интерпретируется как код, который здесь был изначально. Так как в этом коде объявляется новая переменная `b`, он изменяет существующую лексическую область видимости `foo(..)`. Как упоминалось ранее, этот код фактически создает внутри `foo(..)` переменную `b`, которая замещает переменную `b`, объявленную во внешней (глобальной) области видимости.

Когда в программе происходит вызов `console.log(..)`, он находит `a` и `b` в области видимости `foo(..)` и не находит внешнюю переменную `b`. Таким образом, программа выводит «1, 3» вместо «1, 2», как должна была бы.

В этом упрощенном примере передаваемая строка «кода» представляет собой фиксированный литерал. Но она с таким же успехом могла строиться на программном уровне с объединением символов на основании логики программы. Функция `eval(..)` обычно используется для выполнения динамически создаваемого кода, так как динамическое вычисление статического по своей сути кода из строкового литерала не даст никаких реальных преимуществ перед простым написанием кода.

По умолчанию, если строка кода, выполняемая вызовом `eval(..)`, содержит одно или несколько объявлений (переменных или функций), это действие изменяет существующую лексическую область видимости, в которой находится `eval(..)`. С технической точки зрения `eval(..)` может вызываться косвенно с использованием различных трюков (которые выходят за рамки нашего обсуждения), в результате чего она будет выполняться в контексте глобальной области видимости, которая и будет изменяться. Как

бы то ни было, `eval(..)` позволяет изменить лексическую область видимости на стадии выполнения.

При использовании в режиме `strict` `eval(..)` работает в своей собственной лексической области видимости. Это означает, что объявления, созданные внутри `eval(..)`, не будут изменять внешнюю область видимости.

```
function foo(str) {
   "use strict";
   eval( str );
   console.log( a ); // ReferenceError: переменная a
                     // не определена
}

foo( "var a = 2" );
```

В JavaScript существуют и другие средства, приводящие практически к тому же результату, что и `eval(..)`. `setTimeout(..)` и `setInterval(..)` могут получать строку в своем первом аргументе, содержимое которого интерпретируется как код динамически генерируемой функции. Это старое унаследованное поведение, которое давным-давно считается устаревшим. Не используйте его!

Функция-конструктор `new Function(..)` тоже получает в своем последнем аргументе строку кода, который преобразуется в динамически сгенерированную функцию (первый аргумент), если он есть, содержит именованные параметры новой функции). Синтаксис конструктора чуть безопаснее `eval(..)`, но его следует избегать в вашем коде.

Ситуации, требующие динамического генерирования кода в программе, встречаются крайне редко, так как снижение быстродействия почти никогда не стоит того.

with

Другая нежелательная (а теперь считающаяся устаревшей) возможность искажения лексической области видимости в JavaScript основана на использовании ключевого слова `with`. Объяснить происходящее можно несколькими способами, но я предпочитаю объяснять с точки зрения того, как оно взаимодействует с лексической областью видимости и влияет на нее; обычно `with` объясняется как сокращенная запись для многократных обращений к свойствам объекта *без* повторения ссылки на объект.

Пример:

```
var obj = {
    a: 1,
    b: 2,
    c: 3
};
// "рутинные" повторения "obj"
obj.a = 2;
obj.b = 3;
obj.c = 4;
// "более простая" сокращенная запись
with (obj) {
    a = 3;
    b = 4;
    c = 5;
}
```

Тем не менее этот синтаксис представляет собой нечто гораздо большее, чем упрощенную запись для обращения к свойствам объектов. Пример:

```
function foo(obj) {
    with (obj) {
        a = 2;
    }
```

```
}

var o1 = {
    a: 3
};

var o2 = {
    b: 3
};

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

foo( o2 );
console.log( o2.a ); // undefined
console.log( a ); // 2 — ой, утечка глобального значения!
```

В этом примере создаются два объекта, `o1` и `o2`. У одного объекта есть свойство `a`, у другого его нет. Функция `foo(..)` получает ссылку на объект `obj` в аргументе и вызывает `with (obj) { .. }` для этой ссылки. В блоке `with` с переменной `a` встречается то, что выглядит как нормальная лексическая ссылка на переменную `a`, а на самом деле LHS-ссылка (см. главу 1) для присваивания ей значения 2.

При передаче `o1` присваивание `a = 2` находит свойство `o1.a` и присваивает ему значение 2, как показывает следующая команда `console.log(o1.a)`. Однако при передаче `o2`, поскольку у этого объекта нет свойства `a`, такое свойство не создается, и `o2.a` остается неопределенным.

И тут обнаруживается странный побочный эффект: команда `a = 2` создает глобальную переменную `a`. Как такое могло случиться?

Команда `with` получает объект (с 0 или более свойствами) и интерпретирует этот объект так, словно он образует совершенно отдельную лексическую область видимости, а следовательно,

свойства объекта интерпретируются как лексически определенные идентификаторы в этой области видимости.

Несмотря на то что блок `with` интерпретирует объект как лексическую область видимости, для обычного объявления `var` внутри этого блока `with` областью видимости будет не этот блок, а область видимости внешней функции.

Если функция `eval(..)` может изменить существующую лексическую область видимости в том случае, если переданная ей строка кода содержит одно или несколько объявлений, команда `with` создает совершенно новую лексическую область видимости на основе переданного ей объекта.

Если понимать происходящее таким образом, областью видимости, объявленной командой `with` при передаче `o1`, будет `o1`; в этой области видимости существует идентификатор, соответствующий свойству `o1.a`. Но когда в качестве области видимости используется `o2`, идентификатор `a` в ней не обнаруживается, поэтому вступают в действие обычные правила LHS-поиска идентификаторов (см. главу 1).

Ни в области видимости `o2`, ни в области видимости `foo(..)`, ни даже в глобальной области видимости идентификатор `a` найти не удается, поэтому при выполнении команды `a = 2` автоматически создается глобальная переменная (так как программа не выполняется в режиме `strict`).

Мало того что `eval(..)` и `with` использовать не рекомендуется, на них еще и влияет режим `strict`. Команда with запрещена полностью, тогда как различные косвенные или небезопасные формы `eval(..)` запрещены при сохранении базовой функциональности.

Идея превращения объекта и его свойств в область видимости с идентификаторами во время выполнения выглядит довольно странно. Но это самое логичное объяснение для наблюдаемых результатов, которое я могу предложить.

Быстродействие

Как `eval(..)`, так и `with` искажают лексическую область видимости, определенную во время написания программы, посредством изменения или создания новой лексической области видимости во время выполнения.

И что здесь такого, спросите вы? Если они предоставляют более сложную функциональность и гибкость при программировании, разве эти возможности не полезны? Нет.

Движок JavaScript содержит ряд оптимизаций быстродействия, которые выполняются в фазе компиляции. Некоторые из них сводятся к тому, что движок по сути выполняет статический анализ кода в процессе лексического анализа и заранее определяет, где находятся все переменные и объявления функций, чтобы ускорить разрешение идентификаторов в процессе выполнения.

Но если движок обнаруживает в коде `eval(..)` или `with`, он вынужден *предположить*, что вся информация о местонахождении идентификаторов может оказаться недействительной. Ведь во время лексического анализа движок не может знать, какой код может быть передан `eval(..)` для изменения лексической области видимости или что может содержать объект, переданный `with` для создания новой лексической области видимости. Другими словами, в пессимистическом варианте многие эти оптимизации при наличии `eval(..)` или `with` становятся бессмысленными, поэтому движок вообще не выполняет *никаких* оптимизаций.

Ваш код почти наверняка будет работать медленнее просто из-за того, что вы включили в него `eval(..)` или `with`. Как бы умно ни действовал движок в попытках ограничить побочные эффекты этих пессимистичных предположений, **он не сможет обойти тот факт, что без оптимизаций код выполняется медленнее**.

Итоги

Лексическая область видимости определяется решениями о том, где объявляются функции, принимаемыми во время написания программы. Фаза лексического анализа в процессе компиляции располагает информацией о том, где и как объявлены все идентификаторы, — а следовательно, может спрогнозировать их поиск во время выполнения.

В JavaScript существуют два механизма, которые могут «исказить» лексическую область видимости: `eval(..)` и `with`. Первый может изменить существующую лексическую область видимости (во время выполнения), обрабатывая строку «кода» с одним или несколькими объявлениями. Второй фактически создает новую лексическую область видимости (снова во время выполнения), интерпретируя ссылку на объект как область видимости, а свойства этого объекта — как идентификаторы в области видимости.

Недостаток этих механизмов заключается в том, что они не позволяют движку выполнить оптимизации на стадии компиляции, связанные с поиском по областям видимости, потому что движок вынужден пессимистично считать, что такие оптимизации будут недействительными. При использовании любой из этих возможностей программа *будет* работать медленнее. *Не используйте их.*

3 Функциональные и блочные области видимости

Как объяснялось в главе 2, область видимости состоит из пузырей. Каждый пузырь представляет собой нечто вроде контейнера, в котором объявляются идентификаторы (переменные, функции). Пузыри вложены друг в друга, причем это вложение определяется во время написания кода.

Но что именно создает новый пузырь? Только функции? А могут ли другие структуры JavaScript создавать область видимости?

Области видимости из функций

Самый распространенный ответ на эти вопросы — что в JavaScript используются функциональные области видимости. Иначе говоря, каждая объявленная функция создает область видимости для себя, но другие структуры не создают своих областей видимости. Как вы вскоре увидите, это не совсем так.

Но сначала исследуем функциональную область видимости и последствия ее использования. Возьмем следующий пример:

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

    // Какой-то код

    function bar() {
        // ...
    }

    // Еще код

    var c = 3;
}
```

В этом фрагменте пузырь области видимости для `foo(..)` включает идентификаторы `a`, `b`, `c` и `bar`. Неважно, *где* в области видимости находится объявление, в любом случае переменная или функция принадлежит вмещающей ее области видимости. В следующей главе мы более подробно выясним, как это работает.

`bar(..)` имеет собственный пузырь области видимости. Свой пузырь есть и у глобальной области видимости, к которой присоединен всего один идентификатор: `foo`.

Поскольку `a`, `b`, `c` и `bar` принадлежат области видимости `foo(..)`, они недоступны за пределами `foo(..)`. Иначе говоря, в следующем коде произойдут ошибки `ReferenceError`, так как идентификаторы недоступны в глобальной области видимости:

```
bar(); // ошибка

console.log( a, b, c ); // целых 3 ошибки
```

Однако все эти идентификаторы (`a`, `b`, `c`, `foo` и `bar`) доступны внутри `foo(..)`, а также внутри `bar(..)` (предполагается, что в `bar(..)` нет замещающих объявлений идентификаторов).

Функциональная область видимости подчеркивает ту идею, что все переменные принадлежат функции и могут использоваться

во всей этой функции (и даже быть доступными для вложенных областей видимости). Такая архитектура может быть весьма полезной; безусловно, она в полной мере использует «динамическую» природу переменных JavaScript, способных принимать значения разных типов по мере надобности.

С другой стороны, если не принять защитных мер, существование переменных на протяжении всей области видимости может создать непредвиденные проблемы.

Как скрыться у всех на виду

Традиционный подход к функциям подразумевает, что вы объявляете функцию, а потом добавляете в нее код. Однако не менее полезно взглянуть в другом направлении: вы берете произвольный фрагмент кода, написанный вами, и заключаете его в объявление функции, что фактически «скрывает» код от наблюдателя.

На практике вокруг кода создается область видимости, это означает, что любые объявления (переменные или функции) в этом коде теперь будут связаны с областью видимости новой функции-обертки (вместо прежней внешней области видимости). Иначе говоря, можно «скрывать» переменные и функции, заключая их в функциональную область видимости.

Для чего может быть полезно «скрывать» переменные и функции? Существует множество причин для подобной маскировки на базе области видимости. Обычно они происходят от «принципа наименьших привилегий»[1] при проектировании программных продуктов.

Этот принцип гласит, что при проектировании программного продукта (например, API модуля/объекта) следует предоставлять

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

доступ только к тому, что абсолютно необходимо, и «скрывать» все остальное.

Этот принцип распространяется на выбор области видимости, содержащей переменные и функции. Если все переменные и функции будут находиться в глобальной области видимости, конечно, они будут доступны для любой вложенной области видимости. Однако это нарушит «принцип наименьших привилегий», поскольку вы (с большой вероятностью) будете предоставлять доступ ко многим переменным и функциям, которые следовало бы оставить недоступными, так как при правильном использовании кода доступ к этим переменным/функциям будет нежелателен.

Пример:

```
function doSomething(a) {
    b = a + doSomethingElse( a * 2 );
    console.log( b * 3 );
}

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

var b;

doSomething( 2 ); // 15
```

Вероятно, в этом фрагменте переменная `b` и функция `doSomethingElse(..)` относятся к «приватным» подробностям того, как `doSomething(..)` делает свою работу. Предоставление «доступа» к `b` и `doSomethingElse(..)` внешней области видимости не только излишне, но и, скорее всего, опасно в том отношении, что они могут быть использованы непредвиденным образом (намеренно или нет), и это может нарушить предусловия `doSomething(..)`. В более «правильной» архитектуре эти приватные подробности реализации будут скрыты в области видимости `doSomething(..)`, примерно так:

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

    var b;

    b = a + doSomethingElse( a * 2 );

    console.log( b * 3 );
}

doSomething( 2 ); // 15
```

Теперь `b` и `doSomethingElse(..)` недоступны для внешнего влияния, ими распоряжается только `doSomething(..)`. Функциональность и конечный результат не изменились, но архитектура скрывает приватные детали. Такой код обычно считается более качественным.

Предотвращение конфликтов

Другое преимущество «сокрытия» переменных и функций внутри области видимости — предотвращение случайных конфликтов между идентификаторами с одинаковыми именами, но разным предполагаемым использованием. Конфликты часто приводят к непреднамеренной перезаписи значений.

Пример:

```
function foo() {
    function bar(a) {
        i = 3; // изменение `i` в цикле for внешней области
               // видимости
        console.log( a + i );
    }
```

```
    for (var i=0; i<10; i++) {
        bar( i * 2 ); // внимание - бесконечный цикл!
    }
}

foo();
```

Присваивание `i = 3` внутри `bar(..)` неожиданно перезаписывает переменную `i`, объявленную в `foo(..)` в цикле `for`. В данном случае это приведет к бесконечному циклу, потому что `i` присваивается фиксированное значение 3, которое всегда будет оставаться < 10.

Присваивание внутри `bar(..)` должно объявить локальную переменную независимо от выбранного имени идентификатора. `var i = 3;` решит проблему (и создаст упоминавшееся ранее «замещенное» объявление `i`). Дополнительное (не альтернативное) решение — выбор другого имени идентификатора (например, `var j = 3;`). Но возможно, сама структура вашего программного продукта подскажет то же имя идентификатора, так что использование области видимости для «сокрытия» внутреннего объявления — ваш лучший/единственный выход в этой ситуации.

Глобальные пространства имен

Особенно часто конфликты переменных встречаются в глобальной области видимости. Несколько библиотек, загруженных в вашей программе, вполне могут создать конфликты имен, если они не будут должным образом скрывать свои внутренние/приватные функции и переменные.

Такие библиотеки часто создают одно объявление переменной (часто объекта) с достаточно уникальным именем в глобальной области видимости. Этот объект используется как *пространство имен* библиотеки, а весь доступ к функциональности предоставляется через свойства этого объекта вместо идентификаторов верхнего уровня с лексической областью видимости.

Пример:

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

Управление модулями

Другой способ предотвращения конфликтов основан на более современном модульном принципе, использующем один из разнообразных инструментов для управления зависимостями. С такими инструментами никакая библиотека не может добавить идентификаторы в глобальную область видимости; вместо этого она должна явно импортировать свои идентификаторы в другую конкретную область видимости с использованием средств инструмента для управления зависимостями.

Следует заметить, что эти инструменты не обладают «волшебной» функциональностью, избавленной от правил лексической области видимости. Они просто используют правила области видимости так, как объясняется здесь, чтобы гарантировать, что никакие идентификаторы не будут внедряться в общие области видимости, а будут храниться в приватных, не подверженных конфликтам областях видимости, что предотвращает любые случайные конфликты по областям видимости.

При желании вы можете применять защитное программирование и добиться тех же результатов, что и инструменты для управления зависимостями, но без их использования. За дополнительной информацией о модульном паттерне обращайтесь к главе 5.

Функции как области видимости

Как было показано ранее, если взять любой фрагмент кода и «завернуть» его в функцию, вы фактически скроете все внутренние объявления переменных и функций от внешней области видимости во внутренней области видимости этой функции.

Пример:

```
var a = 2;

function foo() { // <-- вставьте это

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

} // <-- и это
foo(); // <-- и это

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

Этот прием работает, но идеальным его не назовешь. Он создает ряд проблем. Во-первых, вам приходится объявлять именованную функцию `foo()`, а значит, имя идентификатора `foo` само «загрязняет» внешнюю область видимости (глобальную в данном случае). Также для выполнения внутреннего кода функцию приходится вызывать по имени.

Было бы удобнее, если бы функция обходилась без имени (а вернее, имя не загрязняло внешнюю область видимости) и при этом могла выполняться автоматически.

К счастью, JavaScript предлагает решение обеих проблем.

```
var a = 2;

(function foo(){ // <-- вставьте это

    var a = 3;
```

```
    console.log( a ); // 3

})(); // <-- и это

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

Давайте разберемся, что здесь происходит.

Сначала обратите внимание на то, что команда `function` для обертки начинается с `(function`... вместо обычного `function`.... На первый взгляд это может показаться второстепенной подробностью реализации, но на самом деле это изменение принципиально. Вместо того чтобы интерпретировать функцию как стандартное объявление, мы интерпретируем ее как функциональное выражение.

Объявление проще всего отличить от выражения по позиции слова `function` в команде (не просто строке, а в отдельной команде). Если команда начинается с `function`, значит, это объявление функции. В противном случае это функциональное выражение.

Ключевое различие между объявлением функции и функциональным выражением относится к тому, где имя связывается с идентификатором.

Сравните два предыдущих фрагмента. В первом фрагменте имя `foo` связывается с внешней областью видимости, и мы вызываем его напрямую `foo()`. Во втором фрагменте имя `foo` не связывается с внешней областью видимости, а только со своей функцией.

Иначе говоря, `(function foo(){ .. })` как выражение означает, что идентификатор `foo` находится только в области видимости, в которой находится `..`, а не во внешней области видимости. Сокрытие имени `foo` означает, что оно не будет без необходимости загрязнять внешнюю область видимости.

Анонимные и именованные функциональные выражения

Вероятно, большинству читателей функциональные выражения знакомы по передаче обратных вызовов в параметрах:

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

Это называется *анонимным функциональным выражением*, потому что `function()`... не содержит идентификатор. Функциональные выражения могут быть анонимными, но в объявлениях функций имя обязательно присутствует, иначе возникнет недопустимая грамматика JS.

Анонимные функциональные выражения удобны и легко вводятся, и многие библиотеки и инструментарии обычно поощряют такой идиоматический стиль программирования. Тем не менее у них есть свои недостатки:

1. Анонимные функции не имеют содержательного имени, которое могло бы выводиться в трассировке стека, что может усложнить отладку.

2. При отсутствии имени, если функция должна ссылаться на саму себя для рекурсии и т. д., приходится использовать устаревшую ссылку `arguments.callee`. Другой пример использования автоссылок — необходимость отсоединения функции-обработчика события после срабатывания.

3. У анонимных функций нет имени, что часто бывает полезно для создания более понятного/удобочитаемого кода. Содержательное имя способствует самодокументированию кода.

Встроенные функциональные выражения мощны и полезны, вопрос анонимности/наличия имени не влияет на это. Предоставление имени для функционального выражения достаточно эффективно решает все эти проблемы и при этом не имеет ощутимых недостатков. Старайтесь всегда задавать имена для своих функциональных выражений:

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

Немедленный вызов функциональных выражений

```
var a = 2;

(function foo(){

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

})();

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

Теперь, когда у нас имеется функция как выражение (для чего она была заключена в пару круглых скобок `()`), мы можем выполнить эту функцию, добавив еще одну пару круглых скобок в конец, например `(function foo(){ .. })()`. Первая пара `()` преобразует функцию в выражение, а вторая пара `()` выполняет функцию.

Этот паттерн настолько распространен, что несколько лет назад сообщество выработало для него специальный термин: *IIFE* (сокращение от «Immediately Invoked Function Expression», то есть «немедленно вызываемое функциональное выражение»).

Конечно, для IIFE имена не обязательны, в самой распространенной форме IIFE используется анонимное функциональное выражение. Именованные IIFE, хотя и встречаются безусловно реже, обладают всеми преимуществами перед анонимными функциональными выражениями, так что эту практику стоит взять на вооружение.

```
var a = 2;

(function IIFE(){

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

})();

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

У традиционной формы IIFE существует небольшая вариация, которую предпочитают некоторые разработчики: `(function() { .. }())`. Внимательно присмотритесь к различиям. В первой форме функциональное выражение заключается в круглые скобки `()`, за которыми размещается вызывающая пара круглых скобок `()`. Во второй форме вызывающие круглые скобки `()` перемещаются внутрь внешней пары `()`. Эти две формы имеют идентичную функциональность. Выбор определяется исключительно стилистическими предпочтениями.

Другая распространенная вариация на тему IIFE — использование того факта, что они по сути являются вызовами функций для передачи аргументов.

Пример:

```
var a = 2;

(function IIFE( global ){
```

```
    var a = 3;
    console.log( a ); // 3
    console.log( global.a ); // 2

})( window );

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

Здесь передается ссылка на объект `window`, но параметру присваивается имя `global`, чтобы в программе существовало четкое стилистическое разграничение между глобальными и неглобальными ссылками. Конечно, вы можете передать все что угодно из внешней области видимости, и параметру можно будет присвоить любое подходящее имя. В основном это чисто стилистический выбор.

Другое применение этого паттерна решает (второстепенную и специфическую) проблему с некорректной перезаписью значений идентификатора `default` по умолчанию, приводящей к неожиданным результатам. Если присвоить параметру имя `undefined`, но не передать значение этого аргумента, можно гарантировать, что идентификатор `undefined` действительно соответствует неопределенному значению в блоке кода:

```
undefined = true; // Создает ловушку для другого кода!
                  // Не делайте так!

(function IIFE( undefined ){

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

})();
```

Еще одна вариация на тему IIFE меняет порядок вещей: выполняемая функция передается в последнюю очередь, после вызова

и передаваемых параметров. Этот паттерн используется в проекте UMD (Universal Module Definition). Некоторые разработчики считают, что он чуть более понятен, хотя и не столь компактен.

```
var a = 2;

(function IIFE( def ){
    def( window );
})(function def( global ){

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

});
```

Функциональное выражение `def` определяется во второй половине фрагмента, а затем передается как параметр (также с именем `def`) функции IIFE, определяемой в первой половине фрагмента. Наконец, параметр `def` (функция) вызывается с передачей `window` как параметра `global`.

Блоки как области видимости

Хотя функции являются наиболее распространенной структурной единицей области видимости и безусловно самым распространенным подходом к разработке большинства JS-кода, находящегося в обращении, возможны и другие структурные единицы области видимости, использование которых может привести к еще более качественному, чистому коду.

Во многих других языках поддерживается блочная область видимости, поэтому разработчики с опытом работы на этих языках привыкли к этой концепции, тогда как для разработчиков, ис-

пользовавших только JavaScript, эта концепция может показаться непривычной.

Но даже если вы не написали ни одной строки кода с расчетом на блочную область видимости, вероятно, вы знакомы с исключительно распространенной идиомой JavaScript:

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

Переменная `i` объявляется непосредственно в заголовке цикла `for`, скорее всего, потому, что мы намерены использовать `i` только в контексте этого цикла `for`. При этом фактически игнорируется тот факт, что область видимости переменной на самом деле распространяется на всю внешнюю область видимости (функциональную или глобальную).

Вся суть блочной области видимости заключается в том, чтобы объявлять переменные как можно ближе к месту их использования, делая их как можно более локальными. Другой пример:

```
var foo = true;

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

Мы используем переменную `bar` только в контексте команды `if`, поэтому в некотором роде логично объявить ее внутри блока `if`. Однако при использовании `var` конкретное место объявления переменных неважно, потому что они всегда принадлежат внешней области видимости. Этот фрагмент фактически имитирует блочную область видимости по стилистическим причинам и полагается на самодисциплину, для того чтобы переменная `bar`

случайно не была использована в другом месте этой области видимости.

Блочная область видимости — инструмент расширения принципа наименьших привилегий от сокрытия информации в функциях до сокрытия информации в блоках кода.

Вернемся к примеру цикла `for`:

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

Зачем загрязнять всю область видимости функции переменной `i`, которая будет (или по крайней мере должна) использоваться только в цикле `for`? Но что еще важнее, разработчики могут предпочесть самостоятельно организовать проверку для защиты от случайного (повторного) использования переменных за пределами их предполагаемой области, например, от получения ошибок об использовании неизвестных переменных при попытке использования их в неправильном месте. Блочная область видимости (если бы она была возможна) для переменной `i` сделает ее доступной только в цикле `for`, что приведет к ошибке при обращении к `i` в любом другом месте функции. Это гарантирует, что переменные не будут повторно использованы так, что это создаст путаницу или усложнит сопровождение кода.

К сожалению, на первый взгляд в JavaScript блочная область видимости не поддерживается.

...Если не присмотреться чуть внимательнее.

with

О `with` было рассказано в главе 2. Хотя к этой конструкции обычно относятся отрицательно, она является примером (разновид-

ности) блочной области видимости — в том отношении, что область видимости, созданная на базе объекта, существует только на протяжении срока жизни этой команды `with`, а не во внешней области видимости.

try/catch

Очень малоизвестный факт: в ES3 было указано, что объявление переменной в секции `catch` конструкции `try/catch` имеет блочную область видимости, ограниченную блоком `catch`.

Пример:

```
try {
    undefined(); // недействительная операция, приводящая
                 // к исключению!
}
catch (err) {
    console.log( err ); // работает!
}
console.log( err ); // ReferenceError: переменная `err`
                    // не найдена
```

Как нетрудно убедиться, `err` существует только в секции `catch`, а при попытке обратиться к переменной в другом месте происходит ошибка.

Хотя это поведение было определено в спецификации и относится практически ко всем стандартным средам JS (кроме, возможно, старых версий IE), многие статические анализаторы кода жалуются, если в одной области видимости присутствуют две и более секции `catch`, объявляющие одноименные переменные ошибок. На самом деле это не является повторным определением, так как переменные имеют блочную область видимости, тем не менее статические анализаторы почему-то жалуются на этот факт.

> Чтобы избежать выдачи ненужных предупреждений, некоторые разработчики присваивают своим catch-переменным имена `err1`, `err2` и т. д. Другие разработчики просто отключают проверку дубликатов имен переменных при статическом анализе.

Может показаться, что природа блочной области видимости секций `catch` не более чем бесполезный теоретический факт, но в приложении Б более подробно рассказано о том, каким полезным он может быть.

let

До сих пор мы видели, что в JavaScript функциональность блочной области видимости предоставляется только странными экзотическими аспектами поведения. Если бы это было все (как это было много, много лет), блочная область видимости была бы практически бесполезной для разработчиков JavaScript.

К счастью, в ES6 ситуация изменилась. Появилось новое ключевое слово `let`, которое используется наряду с `var` для объявления переменных.

Ключевое слово `let` присоединяет объявление переменной к области видимости того блока (обычно паре фигурных скобок `{..}`), в которой оно содержится. Иначе говоря, `let` неявно заимствует область видимости любого блока для объявления своей переменной.

```
var foo = true;

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

```
}

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

Использование `let` для присоединения переменной к существующему блоку происходит не совсем явно. Оно может смутить вас, если вы не будете обращать должного внимания на то, с каким блоками связывается область видимости переменных, и будете часто перемещать блоки, помещать их в другие блоки и т. д. в процессе разработки кода.

Явное создание блоков для блочной области видимости может решить некоторые из этих проблем; разработчик будет четко видеть, к какому блоку присоединяются или не присоединяются переменные. Обычно явный код предпочтительнее неявного или хитроумного. Стиль явного определения блочной области видимости легко реализуется и более естественно соответствует тому, как работают блочные области видимости в других языках:

```
var foo = true;

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

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

Чтобы создать произвольный блок для привязки `let`, достаточно включить пару `{..}` в любое место команды, в котором грамматика допускает размещение команды. В данном случае явный блок создается внутри команды `if`, что может упростить перемещение всего блока при последующем рефакторинге без последствий для позиции и семантики внешней команды `if`.

В главе 4 будет описан механизм *поднятия* (hoisting), при котором объявления интерпретируются как существующие во всей области видимости, в которой они были обнаружены.

Другой способ явного выражения блочной области видимости описан в приложении Б.

Однако объявления с `let` не будут подниматься на всю область видимости того блока, в котором они присутствуют. Такие объявления не будут «существовать» в блоке до команды объявления.

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

Уборка мусора

Другая причина, по которой блочные области видимости полезны в программах, связана с замыканиями и уборкой мусора для освобождения памяти. Здесь я ограничусь кратким описанием, но механизм замыканий подробно объясняется в главе 5.

Возьмем следующий фрагмент:

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

var someReallyBigData = { .. };

process( someReallyBigData );
```

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

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

Обратному вызову функции-обработчика `click` вообще не нужна переменная `someReallyBigData`. Это означает, что теоретически после выполнения `process(..)` большая, занимающая много памяти структура данных может быть уничтожена в ходе уборки мусора. Тем не менее вполне вероятно (хотя и зависит от реализации), что движку JS придется хранить эту структуру, так как функция `click` имеет замыкание над всей областью видимости.

Блочная область видимости может решить эту проблему, более наглядно показывая движку, что хранить `someReallyBigData` не обязательно:

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

// Все объявленное внутри этого блока может пропасть после него!
{
    let someReallyBigData = { .. };
    process( someReallyBigData );
}

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

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

Объявление явных блоков для переменных с целью их локальной привязки — мощный инструмент, который вам стоит добавить в свой инструментарий программирования.

Циклы let

Одна из ситуаций, в которых в полной мере проявляются возможности `let`, — цикл `for`, о чем уже говорилось ранее.

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

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

Ключевое слово `let` в заголовке цикла `for` не только связывает `i` с телом цикла `for` — фактически оно заново выполняет связывание при каждой итерации цикла и обеспечивает повторное присваивание ему значения, существовавшего в конце предыдущей итерации цикла. Поведение связывания при каждой итерации можно продемонстрировать еще и так:

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

Причины, по которым связывание при каждой итерации представляет интерес, станут ясны в главе 5, когда мы перейдем к обсуждению замыканий.

Так как объявления `let` присоединяются к произвольным блокам, а не к области видимости внешней функции (или глобальной области видимости), могут существовать хитроумные ситуации, в которых существующий код скрыто зависит от объявлений `var` с функциональной областью видимости. В таких случаях замена `var` на `let` может потребовать дополнительной осторожности при рефакторинге кода.

Пример:

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

if (foo) {
    var bar = 3;

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

    // ...
}
```

В результате рефакторинга этот код довольно легко приводится к следующему виду:

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

if (foo) {
    var bar = 3;
    // ...
}

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

Но будьте осторожны с такими изменениями при использовании переменных с блочной областью видимости:

```
var foo = true, baz = 10;
if (foo) {
    let bar = 3;
    if (baz > bar) { // <-- не забудьте `bar` при перемещении!
        console.log( baz );
    }
}
```

В приложении Б представлен альтернативный (более явный) стиль блочной области видимости, который может упростить со-

провождение/рефакторинг кода, так как он надежнее ведет себя в таких ситуациях.

const

Кроме `let`, в ES6 появилось ключевое слово `const`, которое также создает константу, «переменную» с блочной областью видимости, значение которой остается фиксированным. Любая попытка изменить это значение в будущем приводит к ошибке.

```
var foo = true;

if (foo) {
    var a = 2;
    const b = 3; // блочная область видимости - внешняя
                 // команда `if`

    a = 3; // нормально!
    b = 4; // ошибка!
}

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

Итоги

Функция — самая распространенная структурная единица области видимости в JavaScript. Переменные и функции, объявленные внутри другой функции, по сути «скрываются» от всех внешних областей — сознательное применение принципа проектирования качественных программных продуктов.

Однако функции никоим образом не являются единственной структурной единицей областей видимости. Под термином «блочная область видимости» понимается концепция принадлежности

переменных и функций произвольному блоку (в общем случае любой паре `{..}`), а не только внешней функции.

Начиная с ES3 структура `try/catch` имеет блочную область видимости в секции `catch`.

В ES6 появилось ключевое слово `let` (родственник ключевого слова `var`), которое позволяет объявлять переменные в произвольном блоке кода. `if (..) { let a = 2; }` объявит переменную, которая фактически заимствует область видимости блока `if's { .. }` и присоединяет себя к ней.

Вопреки мнению некоторых разработчиков, блочную область видимости не следует воспринимать как прямую замену функциональной области видимости `var`. Обе возможности мирно сосуществуют, и разработчики могут и должны использовать как функциональную, так и блочную область видимости там, где это уместно, для создания более качественного, понятного и простого в сопровождении кода.

4 Поднятие

К этому моменту вы должны уже достаточно уверенно понимать концепцию области видимости и присоединения переменных к различным уровням области видимости в зависимости от того, где и как они объявляются. Как функциональная, так и блочная области видимости подчиняются одинаковым правилам в этом отношении: любая переменная, объявленная внутри области видимости, присоединяется к этой области видимости.

Но здесь возникают некоторые нюансы того, как присоединение к областям видимости работает с объявлениями, расположенными в различных точках внутри области видимости. Именно эти нюансы будут рассмотрены в этой главе.

Курица или яйцо?

У разработчика нередко появляется соблазнительная мысль представлять, что весь код программы JavaScript интерпретируется

строка за строкой сверху вниз в ходе выполнения программы. И хотя это в основном верно, в этом предположении есть одна часть, которая может привести к неверным представлениям о вашей программе.

Возьмем следующий код:

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

Как вы думаете, что выведет команда `console.log(..)`?

Многие разработчики ожидают увидеть `undefined`, так как команда `var a` следует за `a = 2`. Вроде бы естественно предположить, что переменная будет определена заново, а следовательно, получит значение `undefined`. Тем не менее программа выведет `2`.

Рассмотрим другой фрагмент кода:

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

Опять-таки появляется соблазн предположить, что поскольку предыдущий фрагмент продемонстрировал отклонение от последовательного выполнения, в этом фрагменте может быть выведено значение `2`. Другие разработчики могут подумать, что поскольку переменная `a` используется перед объявлением, это может привести к выдаче ошибки `ReferenceError`.

К сожалению, обе догадки неверны. Программа выведет `undefined`.

Что здесь происходит? Похоже, мы столкнулись с классической проблемой «курица или яйцо». Что идет первым, объявление («яйцо») или присваивание («курица»)?

Компилятор наносит ответный удар

Чтобы ответить на этот вопрос, необходимо вернуться к главе 1 и нашему обсуждению компиляторов. Вспомните, что движок в действительности компилирует код JavaScript перед его исполнением. В одной из фаз компиляции компилятор находит и связывает все объявления с соответствующими областями видимости. Глава 2 показала, что эта фаза занимает центральное место в лексической области видимости.

Таким образом, лучше всего представить себе, что все объявления переменных и функций обрабатываются сначала, до выполнения любой части вашего кода.

Глядя на команду `var a = 2;`, вы, вероятно, думаете о ней как об одной команде. Но JavaScript в действительности воспринимает ее как две команды: `var a;` и `a = 2;`. Первая команда (объявление) обрабатывается в фазе компиляции. Вторая команда (присваивание) остается на своем месте до фазы исполнения.

Можно считать, что первый фрагмент обрабатывается так:

```
var a;

a = 2;

console.log( a );
```

...где первая часть относится к компиляции, а вторая — к выполнению.

Аналогичным образом второй фрагмент в действительности обрабатывается так:

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

Итак, на этот процесс можно взглянуть так (в метафорическом смысле): объявления переменных и функций «перемещаются» из их текущей позиции в начало кода. При этом происходит *поднятие* (hoisting).

Другими словами, «яйцо» (объявление) следует перед «курицей» (присваивание).

Поднимаются только сами объявления, а все присваивания и другая исполняемая логика остаются на своих местах. Если бы поднятие могло изменять исполняемую логику кода, это породило бы сущий хаос в программе.

```
foo();

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

    var a = 2;
}
```

Объявление функции `foo` (которое в данном случае включает «значение», то есть код функции) поднимается, чтобы вызов в первой строке мог успешно выполниться.

Также важно заметить, что поднятие выполняется на уровне области видимости. Таким образом, хотя наши предыдущие фрагменты были упрощены в том отношении, что они включают только глобальную область видимости, в функции `foo(..)`, которую мы сейчас рассматриваем, переменная `var a` поднимается на верх `foo(..)` (естественно, не на верх самой программы). Таким образом, программу было бы точнее интерпретировать так:

```
function foo() {
    var a;

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

```
    a = 2;
}

foo();
```

Объявления функций поднимаются, как вы только что видели. Функциональные выражения — нет.

```
foo(); // не ReferenceError, но TypeError!
var foo = function bar() {
    // ...
};
```

Идентификатор переменной `foo` поднимается и присоединяется к внешней области видимости (глобальной) программы, так что для `foo()` ошибка `ReferenceError` не происходит. Но у `foo` значения еще нет (как было бы в том случае, если бы это было полноценное объявление функции вместо выражения). Соответственно `foo()` пытается вызвать значение `undefined`, что приводит к недействительной операции `TypeError`.

Также следует напомнить, что хотя это именованное функциональное выражение, идентификатор недоступен во внешней области видимости:

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

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

Этот фрагмент точнее было бы интерпретировать (с поднятием имен) в следующем виде:

```
var foo;

foo(); // TypeError
```

```
bar(); // ReferenceError

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

Сначала функции

Поднимаются как объявления функций, так и объявления переменных. Но здесь есть один нюанс (который может проявиться в коде с несколькими «дубликатами» объявлений): сначала поднимаются функции, а затем переменные.

Пример:

```
foo(); // 1

var foo;

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

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

Вместо 2 выводится 1! Движок интерпретирует этот фрагмент так:

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

foo(); // 1

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

Обратите внимание: `var foo` здесь является дубликатным (а следовательно, игнорируемым) объявлением, хотя оно и предшествует объявлению функции `foo()`..., так как объявления функций поднимаются до нормальных переменных.

Хотя множественные/дубликатные объявления фактически игнорируются, последующие объявления функций *переопределяют* предыдущие.

```
foo(); // 3

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

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

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

Казалось бы, это просто интересная теоретическая информация. Тем не менее она подчеркивает тот факт, что повторяющиеся определения в одной области видимости на самом деле плохая идея, которая часто приводит к странным результатам.

Объявления функций, располагающиеся внутри обычных блоков, обычно поднимаются до вмещающей области видимости (а не являются условными, как подразумевает следующий код):

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

Тем не менее важно заметить, что это поведение ненадежно и может измениться в будущих версиях JavaScript. Вероятно, лучше избегать объявления функций в блоках.

Итоги

Конструкции вида `var a = 2;` было бы заманчиво рассматривать как одну команду, но движок JavaScript рассматривает ее иначе. Он считает `var a` и `a = 2` двумя разными командами: первая обрабатывается во время компиляции, а вторая — во время выполнения.

Все это приводит к тому, что все объявления находятся в области видимости независимо от того, в какой точке они находятся, и обрабатываются заранее *до* выполнения самого кода. Это можно представить себе так, словно объявления (переменных и функций) «перемещаются» вверх своих соответствующих областей видимости; этот процесс называется *поднятием*. Поднимаются сами объявления, но присваивания (и даже присваивания функциональных выражений) *не* поднимаются.

Будьте внимательны с дублирующимися объявлениями, особенно в комбинациях нормальных объявлений и объявлений функций, здесь вас поджидает опасность!

5 Замыкание области видимости

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

А теперь мы обратимся к невероятно важной, но хронически упускаемой из виду, *почти мифологической* части языка: *замыканиям* (closures). Если вы следили за нашим обсуждением лексической области видимости до настоящего момента, замыкания могут показаться вам чем-то банальным, почти разочаровывающим. За волшебным занавесом прячется человек[1], и вы определенно увидите его. Нет, его фамилия не Крокфорд[2]!

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

---

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

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

Просветление

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

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

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

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

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

Технические подробности

Но довольно гипербол и отсылок к фильмам.

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

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

Несколько примеров помогут проиллюстрировать это определение.

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

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

    bar();
}

foo();
```

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

Это и есть замыкание?

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

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

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

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

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

    return bar;
}

var baz = foo();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    bar( baz );
}

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

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

Все эти передачи функций также могут быть косвенными.

```
var fn;

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

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

    fn = baz; // baz присваивается глобальной переменной
}

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

foo();

bar(); // 2
```

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

Теперь я вижу

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

```
function wait(message) {

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

}

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

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

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

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

Замыкание.

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

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

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

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

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

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

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

```
var a = 2;

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

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

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

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

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

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

Я подожду.

Теперь вы увидели!

Циклы и замыкания

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

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

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

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

На самом деле при выполнении этого кода число 6 будет выведено 5 раз с односекундными интервалами.

Что-что?

Сначала объясним, откуда берется 6. Цикл продолжается, пока выполняется условие `i<=5`. В первой итерации, в которой оно не будет выполняться, значение `i` будет равно 6. Итак, в выводе отражено итоговое значение `i` после завершения цикла.

Но если подумать, все становится ясно. Все обратные вызовы тайм-аута прекрасно работают после завершения цикла. Что касается таймеров, даже если бы при каждой итерации использовался вызов `setTimeout(.., 0)`, все эти функции обратного вызова будут выполняться строго после завершения цикла, а следовательно, каждый раз будет выводиться значение 6.

Но здесь возникает более глубокий вопрос. Чего не хватает в нашем коде, чтобы он вел себя так, как мы *подразумевали* с точки зрения семантики?

Мы пытались выразить в коде ту мысль, что каждая итерация цикла «захватывает» собственную копию `i` на момент итерации. Но по правилам работы области действий все пять функций, хотя они и определяются отдельно при каждой итерации цикла, *замыкаются над одной общей глобальной областью видимости*, которая на самом деле содержит только один экземпляр `i`.

Конечно, в таком представлении все функции совместно используют ссылку на одну переменную `i`. В структуре цикла есть что-то такое, что заставляет нас думать, что здесь работают какие-то более глубокие механизмы. Это не так. Все происходит точно так же, как если бы все пять обратных вызовов тайм-аута были объявлены друг за другом, а никакого цикла вообще не было.

Возвращаемся к неотложному вопросу. Чего не хватает? Нужна более замкнутая область видимости. А именно для каждой итера-

ции цикла должна создаваться новая замкнутая область видимости.

Из главы 3 вы узнали, что IIFE создает область видимости, объявляя функцию и немедленно выполняя ее.

Попробуем:

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

Теперь работает? Попробуйте. Я подожду.

Ладно, не буду нагнетать обстановку. Не работает. Но почему? Очевидно, теперь стало больше областей видимости. Каждая функция обратного вызова тайм-аута в самом деле замыкается над своей областью видимости уровня итерации, созданной соответственно каждым выражением IIFE.

Мало создать область видимости для замыкания, если эта область видимости пуста. Присмотритесь внимательнее. Наше выражение IIFE в действительности является пустой, ничего не делающей областью видимости. Чтобы она приносила пользу, в ней должно что-то быть.

Ей нужна своя переменная, чтобы каждая итерация имела собственную копию `i`.

```
for (var i=1; i<=5; i++) {
    (function(){
        var j = i;
        setTimeout( function timer(){
            console.log( j );
```

```
        }, j*1000 );
    })();
}
```

Эврика! Работает!

Небольшая вариация на тему, которая некоторым покажется предпочтительной:

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

Конечно, поскольку IIFE представляют собой всего лишь функции, ничто не мешает нам передать `i`, а потом назвать переменную `j`, потому что нам так удобнее, — и потом снова назвать ее `i`. В любом случае код теперь работает.

Использование IIFE в каждой итерации создает новую область видимости для каждой итерации, что дает функциям обратного вызова тайм-аута возможность создать при каждой итерации замыкание для новой области видимости, которая содержит переменную с правильным для данной итерации значением.

Проблема решена!

Снова о блочной области видимости

Внимательно присмотритесь к нашему анализу предшествующего решения. IIFE используется для создания новой области видимости уровня итерации. Другими словами, нам действительно была *необходима блочная область видимости* уровня итерации. В главе 3

было представлено ключевое слово `let`, которое «перехватывает» блок и объявляет переменную непосредственно в этом блоке.

Фактически блок преобразуется в область видимости, для которой может быть создано замыкание. А это означает, что следующий потрясающий код работает:

```
for (var i=1; i<=5; i++) {
    let j = i; // блочная область видимости для замыкания!
    setTimeout( function timer(){
        console.log( j );
    }, j*1000 );
}
```

(Голосом ведущего телемагазина) *Но и это еще не все!* Для объявлений `let` в заголовке цикла `for` определено специальное поведение. Оно означает, что переменная объявляется не однократно для цикла, а для каждой итерации. И она (для вашего удобства) при каждой последующей итерации будет инициализироваться значением на конец предыдущей итерации.

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

Видали? Блочная область видимости и замыкания работают на пару над решением ваших повседневных задач. Не знаю, как вы, но мне от такого приятнее работать на JavaScript.

Модули

Существуют и другие паттерны программирования, использующие силу замыканий, но внешне вроде бы не имеющие отноше-

ния к обратным вызовам. Рассмотрим самый мощный из них: *модуль*.

```
function foo() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }
}
```

В текущем состоянии этого кода никакие замыкания не видны. Мы просто объявляем приватные переменные `something` и `another`, а также пару внутренних функций `doSomething()` и `doAnother()`; обе имеют лексическую область видимости (а следовательно, и замыкание) над внутренней областью видимости `foo()`.

А теперь посмотрите сюда:

```
function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }

    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
```

```
        doAnother: doAnother
    };
}

var foo = CoolModule();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Этот паттерн JavaScript называется *модулем*. Самый распространенный способ реализации паттерна «модуль» часто называется *предоставлением* (revealing) модуля; именно эта разновидность представлена выше.

А теперь проанализируем некоторые особенности этого кода.

Во-первых, `CoolModule()` — всего лишь функция, но она *должна быть вызвана* для создания экземпляра модуля. Без выполнения внешней функции создание внутренней области видимости и замыканий не произойдет.

Во-вторых, функция `CoolModule()` возвращает объект, обозначенный синтаксисом объектного литерала `{ ключ: значение, … }`. Внутренние функции содержат ссылки на возвращаемый объект, но не на внутренние переменные данных. Эти переменные остаются скрытыми и приватными. По сути, возвращаемый объект может рассматриваться как *открытый программный интерфейс* (API) нашего модуля.

Возвращать фактический объект (литерал) из модуля не обязательно. Также можно вернуть внутреннюю функцию напрямую. jQuery служит хорошим примером такого рода. Идентификаторы jQuery и $ предоставляют открытый API для модуля jQuery, но сами по себе они являются обычными функциями (которые тоже могут обладать свойствами, поскольку все функции являются объектами).

Возвращаемый объект в конечном итоге присваивается внешней переменной `foo`, после чего мы можем обращаться к методам свойств через API, например `foo.doSomething()`.

Функции `doSomething()` и `doAnother()` имеют замыкание над внутренней областью видимости экземпляра модуля (для перехода к которому используется вызов `CoolModule()`). При передаче этих функций за пределы лексической области видимости через обращение к свойствам возвращаемого объекта мы создаем ситуацию, в которой можно наблюдать замыкание и воспользоваться им.

Проще говоря, для использования паттерна «модуль» должны выполняться два требования:

1. Должна существовать внешняя функция-контейнер, которая должна быть вызвана как минимум один раз (каждый раз создается новый экземпляр модуля).

2. Внешняя функция должна возвращать хотя бы одну внутреннюю функцию, чтобы эта внутренняя функция обладала замыканием над приватной областью видимости и могла обращаться и/или изменять это приватное состояние.

Объект со свойством-функцией сам по себе модулем не является. Объект, возвращаемый вызовом функции, который содержит только свойства данных, но не содержит функций, в действительности не является модулем в отношении наблюдаемого поведения.

В предыдущем фрагменте кода представлен автономный создатель модуля с именем `CoolModule()`, который может вызываться любое количество раз; при каждом вызове будет создаваться новый экземпляр модуля. В одной из вариаций на тему этого паттерна должен создаваться только один экземпляр:

```
var foo = (function CoolModule() {
    var something = "cool";
    var another = [1, 2, 3];

    function doSomething() {
        console.log( something );
    }
    function doAnother() {
        console.log( another.join( " ! " ) );
    }

    return {
        doSomething: doSomething,
        doAnother: doAnother
    };
})();

foo.doSomething(); // cool
foo.doAnother(); // 1 ! 2 ! 3
```

Здесь функция модуля была преобразована в выражение IIFE (см. главу 3), мы *немедленно* вызвали его и присвоили возвращенное выражение прямо идентификатору экземпляра единственного модуля `foo`.

Модули представляют собой обычные функции, так что они могут получать параметры:

```
function CoolModule(id) {
    function identify() {
        console.log( id );
    }

    return {
        identify: identify
    };
}

var foo1 = CoolModule( "foo 1" );
```

```
var foo2 = CoolModule( "foo 2" );

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

Другая незначительно отличающаяся, но полезная вариация на тему паттерна «модуль» основана на указании имени объекта, возвращаемого в качестве открытого API:

```
var foo = (function CoolModule(id) {
    function change() {
        // изменение открытого API
        publicAPI.identify = identify2;
    }

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

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

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

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

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

Хранение внутренней ссылки на объект открытого API в экземпляре модуля позволяет изменить этот экземпляр *изнутри*, добавлять и удалять методы и свойства, *а также* изменять их значения.

Современные модули

Различные загрузчики модулей/менеджеры зависимостей фактически упаковывают этот паттерн определения модуля в удобный API. Чтобы вам не изучать одну конкретную библиотеку, позвольте мне представить очень *простую* проверку концепции *для демонстрационных целей* (и только):

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

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

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

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

Ключевая часть этого кода — команда `modules[name] = impl.apply(impl, deps)`. Она вызывает определяющую функцию-обертку для модуля (с передачей любых зависимостей) и сохраняет возвращаемое значение (API модуля) во внутреннем списке модулей, в котором отслеживание осуществляется по имени.

А вот как использовать его для определения нескольких модулей:

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

```
    }

    return {
        hello: hello
    };
} );

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

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

    return {
        awesome: awesome
    };
} );

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

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

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

Модули `"foo"` и `"bar"` определяются функцией, возвращающей открытый API. `"foo"` даже получает экземпляр `"bar"` как параметр зависимости и может использовать его соответственно.

Не жалейте времени и проанализируйте эти фрагменты кода, чтобы в полной мере понять, какую пользу могут принести замыкания для использования в разумных целях. Главный вывод заключается в том, что в менеджерах модулей в действительности

нет никакой магии. Они обладают обеими характеристиками паттерна «модуль», приведенными выше: вызовом определяющей функции-обертки и хранением его возвращаемого значения как API этого модуля.

Другими словами, модули — это всего лишь модули, даже если заключить их в удобную обертку.

Будущие модули

ES6 добавляет полноценную синтаксическую поддержку для концепции модулей. При загрузке через систему модулей ES6 интерпретирует файл как отдельный модуль. Каждый модуль может импортировать другие модули или конкретные составляющие API, а также экспортировать свои открытые составляющие API.

Модули на базе функций не являются статически распознаваемым паттерном (то, что может распознать компилятор); следовательно, семантика их API не учитывается до времени выполнения. А это означает, что вы можете изменить API модуля во время выполнения (см. предшествующее обсуждение открытого API).

С другой стороны, API модулей ES6 статичен (API не изменяется во время выполнения). Так как компилятор знает это, он во время загрузки файла и компиляции может проверить (и проверяет), что ссылка на составляющую API импортированного модуля действительно существует. Если ссылка не существует, компилятор выдает «раннюю» ошибку во время компиляции, не дожидаясь традиционного динамического разрешения на стадии выполнения — и ошибок, если они возникнут.

У модулей ES6 нет «встроенного» формата, они должны определяться в отдельных файлах (по одному на модуль). У браузеров/ядер имеется «загрузчик модулей» по умолчанию (который может переопределяться, но эта тема выходит за рамки нашего обсуждения), который синхронно загружает файл модуля при импортировании.

Пример:

```
bar.js
function hello(who) {
    return "Let me introduce: " + who;
}

export hello;

foo.js
// импортировать только функцию `hello()` из модуля "bar"
import hello from "bar";

var hungry = "hippo";

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

export awesome;

baz.js
// импортировать модули "foo" и "bar" целиком
module foo from "foo";
module bar from "bar";
console.log(
    bar.hello( "rhino" )
); // Let me introduce: rhino

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

Для этого примера необходимо создать отдельные файлы foo.js и bar.js с содержимым из первых двух фрагментов соответственно. Затем программа baz.js должна загрузить/импортировать эти модули для их использования, как показано в третьем фрагменте.

`import` импортирует одну или несколько составляющих из API модуля в текущую область видимости. Каждая составляющая связывается с переменной (`hello` в данном случае). `module` импортирует весь API модуля в связанную переменную (`foo`, `bar` в нашем случае). `exports` экспортирует идентификатор (переменную, функцию) в открытый API текущего модуля. Эти операции могут использоваться сколько угодно раз в определении модуля по мере надобности.

Содержимое *файла модуля* рассматривается так, как если бы оно было заключено в область замыкания области видимости, как и в случае с модулями замыкания функций, представленными ранее.

Итоги

Непросвещенным замыкания кажутся загадочным миром, сокрытым внутри JavaScript, до которого могут добраться только самые отважные. Но на самом деле замыкания — это стандартное и почти очевидное следствие того, как мы пишем код в среде с лексической областью видимости, в которой функции являются значениями и могут произвольно передаваться.

Замыканием называется способность функции запоминать свою лексическую область видимости и обращаться к ней, даже когда функция вызывается за пределами своей лексической области видимости.

Замыкания могут подвести вас (особенно в циклах), если вы не будете узнавать их и понимать, как они работают. Но они также являются невероятно мощным инструментом, который позволяет применять такие паттерны, как *модули*, в различных формах.

Модули должны обладать двумя ключевыми характеристиками: 1) вызываемая внешняя функция-обертка для создания внешней область видимости и 2) возвращаемое значение функции-обертки должно включать ссылку хотя бы на одну внутреннюю функцию, которая обладает замыканием над приватной внутренней областью видимости обертки.

Теперь вы умеете распознавать замыкания в существующем коде и знаете, как обратить их в свою пользу!

Приложение A. Динамическая область видимости

В главе 2 динамическая область видимости сравнивалась на контрасте с моделью лексической области видимости, заложенной в основе работы области видимости в JavaScript (да и в большинстве других языков).

В этом приложении динамическая область видимости рассматривается более подробно, для того чтобы подчеркнуть контраст. Но что еще важнее, динамическая область видимости является близким родственником другого механизма (`this`) в JavaScript, который будет рассматриваться во второй части этой книги.

Как было показано в главе 2, лексическая область видимости представляет собой набор правил, которыми движок руководствуется для поиска переменных и выбора мест, в которых движок их ищет. Ключевая характеристика лексической области видимости — ее определение на стадии написания кода (предполагается, что вы не будете мошенничать с `eval()` или `with`).

Сам термин «динамическая область видимости» наводит на небезосновательную мысль, что существует модель, при которой область видимости может определяться динамически во время

выполнения, а не статически во время написания кода. Это действительно так. Следующий пример демонстрирует этот факт:

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

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

var a = 2;

bar();
```

Лексическая область видимости подразумевает, что RHS-ссылка на `a` в `foo()` будет разрешена в глобальную переменную `a`, в результате чего будет выведено значение `2`.

С другой стороны, динамическая область видимости не обращает внимания на то, как и где объявляются функции и области видимости, важно лишь то, *где они вызываются*. Иначе говоря, цепочка областей видимости основана на стеке вызовов, а не на вложенности областей видимости в коде.

Итак, если бы в JavaScript существовала динамическая область видимости, при выполнении `foo()` теоретически следующий код вывел бы `3`.

```
function foo() {
    console.log( a ); // 3  (не 2!)
}

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

```
var a = 2;

bar();
```

Как такое возможно? Когда `foo()` не удается разрешить ссылку на переменную `a`, вместо перехода вверх по цепочке вложенных (лексических) областей видимости она поднимается вверх по стеку вызовов, чтобы узнать, *откуда была вызвана* функция `foo()`. Так как функция `foo()` была вызвана из `bar()`, она проверяет переменные в области видимости `bar()` и находит ее там со значением 3.

Странно? Наверное, сейчас вам кажется именно так. Но это объясняется тем, что вы, вероятно, работали (или по крайней мере глубоко рассматривали) только над кодом с лексической областью видимости. Таким образом, динамические области видимости могут показаться чем-то противоестественным. Если бы вы с самого начала писали код на языке с динамической областью видимости, это казалось бы вам естественным, а вот лексическая область видимости — чем-то странным.

Чтобы было понятно: в JavaScript нет динамической области видимости. В языке используется лексическая область видимости. Просто и ясно. Тем не менее механизм `this` отчасти напоминает динамическую область видимости.

Главное отличие: лексическая область видимости определяется во время написания кода, а динамическая область видимости (и `this`!) определяется во время выполнения. Для лексической области видимости важно то, где функция была объявлена, а для динамической — где она была вызвана.

Наконец, для `this` важно то, как функция была вызвана; это показывает, как тесно связан этот механизм с идеей динамической области видимости. Дополнительную информацию можно найти во второй части этой книги.

Приложение Б. Полифилы для блочной области видимости

В главе 3 исследовалась блочная область видимости. Было показано, что `with` и секция `catch` — всего лишь маленькие примеры блочной области видимости, которая существовала в JavaScript по крайней мере с появления ES3.

Но только с появлением `let` в ES6 в программах наконец-то появилась полноценная, ничем не ограниченная поддержка блочной области видимости в программах. Блочная область видимости позволяет сделать много интересного как на уровне функциональности, так и на стилистическом уровне. Но что, если вам захочется использовать блочные области видимости в средах до ES6?

Возьмем следующий код:

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

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

Он отлично работает в средах ES6. А если нужно сделать то же в более ранней среде? На помощь приходит `catch`.

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

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

Что за уродливый и нелепый код! Команда `try/catch` вроде бы принудительно инициирует ошибку, но эта «ошибка» обычное значение 2, а объявление переменной, которая получает ее, находится в секции `catch(a)`. Полный абсурд.

Да, секция `catch` действительно обладает блочной областью видимости, а следовательно, может использоваться в качестве полифила для блочной области видимости в средах до ES6.

«Кому захочется писать такой уродливый код?» — спросите вы. Верно. Код, выводимый компилятором CoffeeScript, тоже никто не пишет (по крайней мере его часть). Дело не в этом.

Существуют инструменты, транспилирующие код ES6 для работы в средах до ES6. Вы можете писать код, используя блочную область видимости, и пользоваться этой функциональностью. Инструментарий фазы сборки возьмет на себя рутинное преобразование и сгенерирует код, который будет *работать* при развертывании.

На самом деле именно так выглядит предпочтительный путь миграции для всех (или почти всех) новых возможностей ES6: транспилятор используется для того, чтобы на основе кода ES6 сгенерировать ES5-совместимый код на время перехода с более ранних сред на ES6.

Traceur

Компания Google ведет проект Traceur[1], целью которого является транспиляция возможностей ES6 в более ранние среды (в основном ES5, но не только) для общего использования. Комитет TC39 использует этот инструмент (и другие) для тестирования семантики новых возможностей, которые он разрабатывает.

Какой код Traceur сгенерирует для этого фрагмента? Наверное, вы уже догадались!

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

console.log( a );
```

Итак, с такими инструментами вы можете начать пользоваться блочной видимостью независимо от того, ориентируетесь вы на ES6 или нет, потому что конструкция `try/catch` существует (и работает именно так) еще со времен ES3.

Неявные и явные блоки

В главе 3 выявляются некоторые потенциальные проблемы с удобством сопровождения/рефакторинга при введении блочной области видимости. Существует ли другой способ использования блочной области видимости, лишенный этих проблем?

[1] http://traceur-compiler.googlecode.com/git/demo/repl.html.

Рассмотрим следующую альтернативную форму `let`, которая называется блоком `let` или командой `let` (в отличие от рассмотренных ранее объявлений `let`):

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

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

Вместо того чтобы неявно перехватывать существующий блок, команда `let` создает явный блок для своего связывания области видимости. Явный блок не только сильнее выделяется (и вероятно, более надежно переносит рефакторинг кода), он создает код, более понятный на уровне грамматики, принудительно смещая все объявления в начале блока. Так вам будет проще при взгляде на любой блок понять, что относится или не относится к его области видимости.

Как паттерн он отражает подход, используемый многими разработчиками с функциональной областью видимости, когда они вручную перемещают/поднимают все объявления `var` в начало своих функций. Команда `let` сознательно помещает их в начало блока, и, если вы не используете объявления `let`, разбросанные по всему коду, ваши объявления с блочной областью видимости лучше видны и создают меньше проблем с сопровождением.

Но здесь возникает проблема: форма команды `let` не включена в ES6. Официальный компилятор Traceur не принимает этот код.

Есть два возможных решения: отформатировать код в действительном синтаксисе ES6 с небольшой долей дисциплины программирования:

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

```
}

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

Возможен и другой вариант: инструменты создаются для решения наших проблем. Таким образом, другой вариант — написать явные блоки с командами `let`, чтобы инструментарий преобразовал их в действительный, работоспособный код.

Для решения этой проблемы я написал утилиту let-er[1]. Let-er — транспилятор кода, работающий на стадии сборки, но его единственной задачей является поиск форм команд `let` и их транспиляция. Весь остальной код остается без изменений, включая любые объявления `let`. Вы можете безопасно использовать let-er на первом шаге транспиляции ES6, а затем при необходимости обработать свой код таким инструментом, как Traceur.

У let-er имеется флаг конфигурации `--es6`, при установке которого (по умолчанию он сброшен) изменяется генерируемый код. Вместо трюка с полифилом `try/catch` для ES4 let-er берет фрагмент и генерирует полностью ES6-совместимый код без всяких трюков:

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

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

Итак, вы можете немедленно начать пользоваться let-er и писать код, предназначенный для любых сред, предшествующих ES6. А когда вас будет интересовать только ES6, добавьте флаг — и ваш код мгновенно начнет генерироваться только для ES6.

[1] https://github.com/getify/let-er.

А самое важное — вы можете использовать предпочтительную и явно выраженную форму команды `let` даже при том, что она еще не является официальной частью какой-либо версии ES (пока).

Быстродействие

Позвольте мне добавить одну быструю заметку по быстродействию `try/catch`, а также ответить на вопрос: «Почему бы просто не использовать IIFE для создания области видимости?»

Во-первых, быстродействие `try/catch` ниже, но нет разумных оснований считать, что это неизбежно или что всегда будет именно так. С того момента, когда транспилятор ES6, официально утвержденный TC39, использует `try/catch`, команда Traceur предложила Chrome повысить быстродействие `try/catch`, и, разумеется, это предложение было услышано.

Во-вторых, IIFE трудно напрямую сравнивать с `try/catch`, потому что функция, в которую обернут любой произвольный код, изменяет смысл `this`, `return`, `break` и `continue` внутри этого кода. IIFE не может считаться подходящей заменой общего плана. Эта конструкция может использоваться только вручную в некоторых случаях.

Вопрос следует ставить иначе: нужна вам блочная область видимости или нет? Если нужна, то эти средства предоставят вам такую возможность. Если нет, то продолжайте использовать `var` и спокойно программируйте!

Приложение B. Лексическое this

Хотя название не дает сколько-нибудь подробной информации об этом механизме, в ES6 есть одна тема, связывающая `this` с лексической областью видимости. Мы рассмотрим ее в общих чертах.

В ES6 появилась специальная синтаксическая форма объявления функций — так называемые *стрелочные функции*. Она выглядит так:

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

foo( 2 ); // 2
```

Так называемая «толстая стрелка» часто ставится в пример как сокращенная запись для *раздражающе громоздкого* (сарказм) ключевого слова `function`.

Но со стрелочными функциями связано нечто гораздо более важное, что не имеет никакого отношения к сокращению объявлений на несколько нажатий клавиш. Например, в следующем коде существует проблема:

```
var obj = {
    id: "awesome",
    cool: function coolFn() {
```

```
        console.log( this.id );
    }
};

var id = "not awesome";

obj.cool(); // awesome

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

Проблема — потеря связывания `this` с функцией `cool()`. У этой проблемы есть несколько решений, но одно из часто встречающихся решений выглядит так: `var self = this;`.

Это может выглядеть так:

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

        if (self.count < 1) {
            setTimeout( function timer(){
                self.count++;
                console.log( "awesome?" );
            }, 100 );
        }
    }
};

obj.cool(); // awesome?
```

Не углубляясь в дебри, скажу, что «решение» `var self = this` просто обходит всю проблему понимания и правильного использования этого связывания, а вместо этого возвращается к тому, что, возможно, кажется разработчику более комфортным: к лексической области видимости. `self` становится обычным идентификатором, который может разрешаться через лексическую область видимости и замыкания, и при этом не обращает ни

малейшего внимания на то, что при этом происходит со ссылкой `this`.

Людям не нравится писать пространные тексты, особенно когда это приходится делать снова и снова. Спецификация ES6 должна была исправить ситуацию и решить некоторые распространенные проблемы с идиомами, такими как эта. Решение ES6 — стрелочная функция — вводит новое поведение, называемое *лексическим this*.

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

obj.cool(); // awesome?
```

Вкратце скажу, что в отношении связывания `this` стрелочные функции ведут себя совсем не так, как обычные функции. Они отбрасывают все нормальные правила связывания `this` и используют вместо них значение `this` непосредственной лексической внешней области видимости, какой бы она ни была.

Итак, в этом фрагменте стрелочная функция не получает значение `this`, связанное каким-то непредсказуемым образом; она просто «наследует» привязку `this` функции `cool()` (и это правильно, если функция вызывается так, как показано).

Хотя этот синтаксис способствует созданию более компактного кода, на мой взгляд, стрелочные функции только внедряют в син-

таксис типичную ошибку разработчиков, которые путают и объединяют правила связывания `this` с правилами лексической области видимости.

Другими словами, стоит ли прикладывать массу усилий и мириться с громоздкостью парадигмы программирования `this` только для того, чтобы тут же ослабить ее возможности, смешивая ее с лексическими ссылками? Более естественно выбрать тот или иной подход для отдельного фрагмента кода и не смешивать в нем разные подходы.

Еще один недостаток стрелочных функций — их анонимность. Причины, по которым именованные функции предпочтительнее анонимных, изложены в главе 3.

Правильный подход к «проблеме `this`» — понимание и правильное использование механизма `this`.

```
var obj = {
    count: 0,
    cool: function coolFn() {
        if (this.count < 1) {
            setTimeout( function timer(){
                this.count++; // `this` безопасно
                              // благодаря `bind(..)`
                console.log( "more awesome" );
            }.bind( this ), 100 ); // look, `bind()`!
        }
    }
};

obj.cool(); // more awesome
```

Какой бы вариант вы ни предпочли — новое лексическое поведение `this` стрелочных функций или проверенное време-

нем поведение `bind()`, — важно заметить, что стрелочные функции не только избавляют вас от необходимости вводить слово `function`.

У них есть *сознательно спроектированное отличие в поведении*, которое следует изучить и понять, и, если вы так решите, — использовать.

Теперь, когда вы в полной мере понимаете лексическую область видимости и замыкание, разобраться в лексическом поведении `this` будет проще простого!

ЧАСТЬ 2

this и прототипы объектов

Предисловие

Пока я читал эту книгу, готовясь к написанию предисловия ко второй части, мне пришлось размышлять над тем, как я изучал JavaScript и насколько язык изменился за последние 15 лет, в течение которых я программировал и занимался разработкой.

Когда я начал работать на JavaScript 15 лет назад, практика использования в веб-страницах технологий, не относящихся к HTML (таких, как CSS и JS), называлась DHTML (или Dynamic HTML). В те времена JavaScript в основном использовался для добавления анимированных снежинок или динамических часов, которые отображали текущее время в строке состояния. Достаточно сказать, что я не обращал особого внимания на JavaScript на ранней стадии своей карьеры из-за необычности реализаций, которые я часто находил в интернете.

Только в 2005 году я впервые заново открыл для себя JS как настоящий язык программирования, который заслуживает более пристального внимания. После изучения первой бета-версии Google Maps я осознал его потенциал. На тот момент приложение Google Maps было уникальным — оно позволяло перемещаться по карте при помощи мыши, менять масштаб и выдавать запросы к серверу без перезагрузки страницы. И все это из кода JavaScript. Происходящее казалось волшебным!

А когда что-то кажется таковым, обычно это верный признак того, что вы стоите на пороге нового технологического прорыва. Предчувствия не обманули: сегодня я могу сказать, что JavaScript — это один из основных языков, которые я использую для программирования на стороне клиента и сервере, и я его ни на что не променяю.

Если я о чем-то и жалею, так разве что о том, что не уделил JavaScript большего внимания до 2005 года. Точнее, что мне не хватило проницательности, чтобы рассмотреть в JavaScript полноценный язык программирования, который может приносить не меньше пользы, чем C++, C#, Java и многие другие.

Если бы серия книг «Вы не знаете JS» оказалась у меня в самом начале карьеры, моя история могла бы сильно отличаться от нынешней. И это одна из особенностей серии, которые мне особенно нравятся: JavaScript объясняется на уровне, который формирует ваше понимание в процессе чтения, но делает это интересно и содержательно.

Вторая часть расширяет знания о замыканиях для очень важной части языка JS — ключевого слова `this` и прототипов. Эти две простые вещи имеют крайне важное значение для того, что вы узнаете в будущих книгах, потому что они закладывают фундамент для полноценного программирования на JavaScript. Концепции создания объектов, установления связей между ними и расширения для представления различных сущностей вашего приложения необходимы для создания больших и сложных приложений в JavaScript. Без них мы бы не смогли создавать сложные приложения (такие, как Google Maps) на JavaScript.

Вероятно, подавляющее большинство веб-разработчиков никогда не создает объекты JavaScript и просто рассматривает язык как «клей» для привязки событий между кнопками и запросами AJAX. Одно время я тоже принадлежал к этому лагерю, но когда я узнал,

как строить прототипы и создавать объекты в JavaScript, передо мной открылся целый мир новых возможностей. Если вы относитесь к категории «простых создателей связующего кода», эта книга обязательна к прочтению; если же вам достаточно освежить память, то книга поможет вам и в этом. В любом случае вы не будете разочарованы. Не сомневайтесь!

Ник Берарди (Nick Berardi)
nickberardi.com, @nberardi

6 Что такое this?

> Любая достаточно *развитая* технология неотличима от магии.
>
> *Артур Кларк*

Одним из самых запутанных механизмов в JavaScript является ключевое слово `this`. Оно обозначает специальный идентификатор, который автоматически определяется в области видимости каждой функции, но вопрос о том, на что именно он ссылается, может ввести в заблуждение даже самых опытных разработчиков JavaScript.

Механизм `this` в JavaScript на самом деле *не настолько* развит, но разработчики часто перефразируют эту цитату, вставляя в нее «сложная», «запутанная» или что-нибудь в этом роде. А впрочем, без ясного понимания происходящее вполне можно принять за магию.

Для чего нужно this?

Если механизм `this` настолько запутан даже для опытных разработчиков JavaScript, то есть ли от него реальная польза? Не со-

здает ли он больше проблем, чем решает? Прежде чем заняться вопросом «как», нужно изучить вопрос «почему».

Следующий пример демонстрирует необходимость и пользу `this`:

```
function identify() {
    return this.name.toUpperCase();
}

function speak() {
    var greeting = "Hello, I'm " + identify.call( this );
    console.log( greeting );
}

var me = {
    name: "Kyle"
};

var you = {
    name: "Reader"
};

identify.call( me ); // KYLE
identify.call( you ); // READER

speak.call( me ); // Hello, I'm KYLE
speak.call( you ); // Hello, I'm READER
```

Если вопрос «как» в этом фрагменте сбивает вас с толку — не огорчайтесь! Вскоре мы доберемся до этого. Просто ненадолго отложите эти вопросы, чтобы мы могли более полно разобраться с вопросом «зачем».

Этот фрагмент кода позволяет повторно использовать функции `identify()` и `speak()` для разных *контекстных* объектов (`me` и `you`), вместо того чтобы требовать создания отдельной версии функции для каждого объекта. Вместо того чтобы полагаться на `this`, можно было явно передать контекстный объект `identify()` и `speak()`:

```
function identify(context) {
    return context.name.toUpperCase();
}

function speak(context) {
    var greeting = "Hello, I'm " + identify( context );
    console.log( greeting );
}

identify( you ); // READER
speak( me ); // Hello, I'm KYLE
```

Тем не менее механизм `this` предоставляет более элегантный механизм неявной «передачи» ссылки на объект, что приводит к более ясной архитектуре API и простоте повторного использования.

Чем сложнее ситуация, в которой вы используете это решение, тем четче видно, что передача контекста в явном параметре часто оказывается более громоздкой, чем передача контекста `this`. При более близком анализе объектов и прототипов можно увидеть, насколько удобно, когда коллекция функций может автоматически обращаться к подходящему контекстному объекту.

Путаница

Вскоре мы объясним, как работает ключевое слово `this`, но сначала нужно развеять некоторые недоразумения относительно того, как `this` *не* работает.

Имя `this` создает некоторую путаницу, когда разработчики пытаются интерпретировать его слишком буквально. Часто подразумеваются два значения, но оба ошибочны.

Сама функция

Первое распространенное заблуждение — считать, что `this` относится к самой функции. Кстати, такое заключение как минимум разумно с точки зрения грамматики.

Для чего ссылаться на функцию из самой функции? Самые распространенные причины — такие конструкции, как рекурсия (вызов функции из самой себя) или создание обработчика ошибок, которые могут прерывать свое связывание при первом вызове.

Разработчики, не имеющие опыта с механизмами JavaScript, часто думают, что ссылка на функцию как на объект (все функции в JavaScript являются объектами) позволяет сохранять состояние (значения свойств) между вызовами. Хотя это безусловно возможно и имеет некоторое ограниченное применение, в книге будет представлено много других паттернов с более удобными хранилищами для состояния помимо объектов функций.

Но пока мы исследуем этот паттерн и убедимся в том, что `this` не позволяет функции получить ссылку на саму себя, как вы, возможно, предполагали.

Рассмотрим следующий код, в котором мы пытаемся подсчитать, сколько раз была вызвана функция (`foo`):

```
function foo(num) {
    console.log( "foo: " + num );

    // Подсчет вызовов `foo`
    this.count++;
}

foo.count = 0;

var i;
```

```
for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз была вызвана функция `foo`?
console.log( foo.count ); // 0 -- ЧТО?!
```

Значение `foo.count` все равно остается равным 0, хотя четыре команды `console.log` четко показывают, что функция `foo(..)` была вызвана четыре раза. Недоразумение происходит от слишком буквальной интерпретации смысла `this` (в `this.count++`).

Когда в коде выполняется команда `foo.count = 0`, она действительно добавляет свойство `count` к объекту функции `foo`. Но ссылка `this.count` внутри функции на самом деле указывает *вовсе не на объект функции*, так что даже несмотря на одинаковые имена свойств, корневые объекты различаются; так возникает путаница.

Ответственный разработчик в этот момент должен спросить: «Если я увеличиваю свойство count, но не то, которое предполагалось, какое же свойство count я увеличиваю?» При более глубоком анализе выясняется, что он случайно создал глобальную переменную count (если вас интересует, как это произошло, см. главу 7), и в настоящее время переменная имеет значение `NaN`. Конечно, после обнаружения этого странного результата появляется целый ряд других вопросов: «Почему переменная оказалась глобальной и почему она содержит `NaN` вместо какого-нибудь конкретного значения?»

Вместо того чтобы остановиться на этом месте и начать выяснять, почему ссылка `this` не работает так, как ожидалось, а также искать ответы на эти трудные, но важные вопросы, многие разработчики просто держатся подальше и пытаются найти другое решение, например, создать другой объект для хранения свойства `count`:

```
function foo(num) {
    console.log( "foo: " + num );

    // Подсчет вызовов `foo`
    data.count++;
}

var data = {
    count: 0
};

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз была вызвана функция `foo`?
console.log( data.count ); // 4
```

Хотя этот подход «решает» проблему, к сожалению, он просто игнорирует реальную проблему (плохое понимание того, что это означает и как работает), а лишь возвращается в зону комфорта более знакомого механизма: лексической области видимости.

Лексическая область видимости — прекрасный, полезный механизм; я нисколько не пытаюсь умалить его достоинства. Но если вы постоянно гадаете, как использовать `this`, и так же постоянно ошибаетесь — это еще не значит, что вам остается лишь отступить к лексической области видимости, так и не узнав, чего же вы не поняли.

Чтобы сослаться на объект функции из нее самой, одного ключевого слова `this` обычно недостаточно. В общем случае нужно получить ссылку на объект функции через лексический идентификатор (переменную), указывающий на нее.

Рассмотрим эти две функции:

```
function foo() {

    foo.count = 4; // `foo` ссылается на себя
}

setTimeout( function(){
    // анонимная функция (без имени), не может ссылаться на себя
}, 10 );
```

В первой функции — *именованной* — `foo` является ссылкой, которая может использоваться для обращения к функции из нее самой.

Но во втором примере функция обратного вызова, передаваемая `setTimeout(..)`, не имеет идентификатора (*анонимная функция*), поэтому нормального способа обратиться к объекту функции не существует.

Старая, а ныне устаревшая и нежелательная ссылка `arguments.callee` внутри функции также ссылается на объект функции, выполняемой в настоящий момент. Эта ссылка обычно дает единственную возможность обратиться к объекту анонимной функции из нее самой. Тем не менее лучшее решение — вообще обходиться без анонимных функций (по крайней мере тех,

которые должны обращаться к самим себе) и использовать именованные функции (выражения). Конструкция `arguments.callee` устарела, и использовать ее не рекомендуется.

Таким образом, в нашем примере можно было бы повсюду использовать идентификатор `foo` как ссылку на объект функции и вообще обойтись без `this`:

```
function foo(num) {
    console.log( "foo: " + num );

    // Подсчет вызовов `foo`
    foo.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        foo( i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз была вызвана функция `foo`?
console.log( foo.count ); // 4
```

Однако и этот подход обходит проблему без *реального* понимания `this` и полностью полагается на лексическую область видимости переменной `foo`.

Другое возможное решение — заставить `this` действительно указывать на объект функции `foo`:

```
function foo(num) {
    console.log( "foo: " + num );

    // Подсчет вызовов `foo`.
    // Примечание: `this` в действительности сейчас содержит
    // `foo` из-за того, как была вызвана функция `foo`
    // (см. ниже)
    this.count++;
}

foo.count = 0;

var i;

for (i=0; i<10; i++) {
    if (i > 5) {
        // используя `call(..)`, мы гарантируем, что `this`
        // указывает на сам объект функции (`foo`)
        foo.call( foo, i );
    }
}
// foo: 6
// foo: 7
// foo: 8
// foo: 9

// сколько раз была вызвана функция `foo`?
console.log( foo.count ); // 4
```

Вместо того чтобы избегать `this`, мы приветствуем его. Вскоре мы более подробно объясним, как работают такие механизмы, так что если вы чего-то не понимаете, не беспокойтесь!

Область видимости

Следующее распространенное заблуждение о `this` — то, что оно каким-то образом связано с областью видимости функции. Это тонкий вопрос, потому что в одном смысле здесь есть доля правды, но в другом оно совершенно ошибочно.

Внесу ясность: this ни в каком отношении не указывает на лексическую область видимости функции. Действительно, во внутреннем представлении область видимости может рассматриваться как объект, содержащий свойство для каждого из доступных идентификаторов. Однако «объект» области видимости недоступен для кода JavaScript. Он является внутренней частью реализации *движка*.

Возьмем код, который пытается (безуспешно) переступить границу и использовать this для неявного обозначения лексической области видимости функции:

```
function foo() {
    var a = 2;
    this.bar();
}

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

foo(); //ReferenceError: значение a не определено
```

Этот фрагмент содержит сразу несколько ошибок. Хотя на первый взгляд он кажется неестественным, приведенный код представляет собой выжимку из реального кода, который встречается на общедоступных форумах поддержки сообщества. Это отличная (хотя и печальная) иллюстрация того, насколько ошибочными бывают предположения относительно this.

Первая ошибка — это попытка обратиться к функции bar() через this.bar(). То, что она работает, — фактически чистая случайность, но вскоре мы поговорим о том, *как* это происходит. Самый естественный способ вызова bar() — простая лексическая ссылка по идентификатору без использования префикса this.

Разработчик, который пишет такой код, пытается использовать `this` для того, чтобы проложить мост между лексическими областями видимости `foo()` и `bar()`, чтобы предоставить `bar()` доступ к переменной `a` во внутренней области видимости `foo()`. Такие мосты невозможны. Вы не сможете использовать ссылку `this` для поиска чего-то в лексической области видимости. Это невозможно.

Каждый раз, когда вам захочется смешать поиск по лексической области видимости с `this`, напомните себе: *такого моста не существует*.

Что такое this?

Разобравшись с различными неверными предположениями, обратимся к тому, как на самом деле работает механизм `this`.

Ранее мы сказали, что связывание `this` осуществляется не во время написания программы, а во время выполнения. Связывание является контекстным, то есть базируется на условиях вызова функции. Связывание `this` не имеет никакого отношения к тому, где была объявлена функция, но полностью определяется способом вызова этой функции.

При вызове функции создается *запись активации*, также называемая *контекстом выполнения*. Запись содержит информацию о том, откуда была вызвана функция (стек вызовов), как она была вызвана, какие параметры были переданы при вызове и т. д. Одним из свойств этой записи является ссылка `this`, которая используется на протяжении выполнения этой функции.

В следующей главе вы узнаете, как найти *место вызова* (call-site) функции для определения того, как при ее выполнении будет производиться связывание `this`.

Итоги

Связывание `this` — постоянный источник путаницы для разработчиков JavaScript, которые не удосужились выяснить, как же в действительности работает механизм. Догадки, пробы и ошибки, тупое копирование кода из ответов на сайте Stack Overflow нельзя назвать эффективным или правильным способом использования важного механизма `this`.

Чтобы изучить `this`, необходимо сначала узнать, чем `this` *не* является (какие бы предположения или заблуждения ни привели вас к этим путям). `this` не является ссылкой на саму функцию и не является ссылкой на *лексическую* область видимости функции.

Связывание `this` происходит в момент вызова функции, и то, на *что* ссылается `this`, определяется исключительно местом вызова, из которого была вызвана функция.

7 this обретает смысл!

В предыдущей главе мы отбросили различные неверные представления по поводу `this` и узнали, что связывание `this` осуществляется для каждого вызова функции на основании исключительно места вызова (то есть способа вызова функции).

Место вызова

Чтобы понять связывание, необходимо понять концепцию *места вызова* (call-site): позицию кода, в которой вызывается функция (а не ту, где она была объявлена). Необходимо проанализировать место вызова, чтобы ответить на вопрос: на что указывает ссылка `this`?

Поиск места вызова в общем случае означает «найти точку, из которой вызывается функция», что порой может оказаться непросто, так как некоторые паттерны программирования могут скрыть истинное место вызова.

При этом важно учитывать стек вызовов (то есть стек функций, которые были вызваны для того, чтобы выполнение дошло до сво-

его текущего состояния). Место вызова, которое нас интересует, — вызов непосредственно перед текущей выполняемой функцией.

А теперь продемонстрируем концепции стека вызовов и точки вызова:

```
function baz() {
    // Стек вызовов: `baz`
    // Следовательно, место вызова принадлежит глобальной
    // области видимости

    console.log( "baz" );
    bar(); // <-- Место вызова для `bar`
}

function bar() {
    // Стек вызовов: `baz` -> `bar`
    // Следовательно, место вызова находится в `baz`
    console.log( "bar" );
    foo(); // <-- Место вызова для `foo`
}

function foo() {
    // Стек вызовов: `baz` -> `bar` -> `foo`
    // Следовательно, место вызова находится в `bar`
    console.log( "foo" );
}

baz(); // <-- Место вызова для `baz`
```

Анализируя код для поиска фактического места вызова (из стека вызовов), будьте внимательны, потому что это единственное, что влияет на связывание `this`.

Чтобы мысленно представить стек вызовов, можно последовательно рассмотреть цепочку вызовов функций, как это было сделано в комментариях в предыдущем примере. Тем не менее такой подход утомителен и чреват ошибками. Другой способ просмотра стека вызовов основан на использовании отладчика

в браузере. В большинстве современных браузеров для настольных систем имеются встроенные средства разработчика, включающие отладчик JS. В предыдущем фрагменте кода можно установить в отладчике точку прерывания в первой строке функции `foo()` или просто вставить в этой первой строке команду `debugger;`. При запуске страницы отладчик прервет выполнение в этой позиции и выведет список функций, которые были вызваны для перехода к этой строке, — это и будет стек вызовов. Итак, если вы пытаетесь диагностировать проблемы в связывании `this`, воспользуйтесь средствами разработчика для получения стека вызовов, а потом найдите второй элемент сверху — в нем будет приведено настоящее место вызова.

Ничего кроме правил

А теперь посмотрим, как место вызова определяет, на что будет указывать `this` в процессе выполнения функции.

Проанализируйте место вызова и определите, какое из четырех правил применяется в конкретном случае. Сначала мы объясним каждое из четырех правил по отдельности, а затем продемонстрируем их приоритеты, если к месту вызова *можно* применить сразу несколько правил.

Связывание по умолчанию

Первое правило, которое мы проанализируем, происходит от самого распространенного случая вызовов функций: *автономных вызовов функций*. Считайте, что это правило будет использовано по умолчанию, если не применяется ни одно из других правил.

Рассмотрим следующий код:

```
function foo() {
    console.log( this.a );
```

```
}

var a = 2;

foo(); // 2
```

Первое, на что следует обратить внимание, — на переменные, объявленные в глобальной области видимости (например, `var a = 2`): они имеют синонимы в виде одноименных свойств глобального объекта. Они не являются независимыми копиями, это просто одно и то же, что-то вроде двух сторон одной монеты.

Кроме того, мы видим, что при вызове `foo()` конструкция `this.a` превращается в глобальную переменную `a`. Почему? Потому, что в этом случае *связывание по умолчанию* `this` применяется к вызову функции, а следовательно, `this` указывает на глобальный объект.

Откуда мы узнали, что здесь применяется правило связывания по умолчанию? Мы анализируем место вызова и смотрим, как вызывается `foo()`. В нашем фрагменте `foo()` вызывается с простой, неизмененной ссылкой на функцию. Никакие другие правила (которые будут приведены ниже) здесь не применимы, поэтому в данном случае применяется связывание по умолчанию.

Если действует режим `strict`, глобальный объект не может использоваться для связывания по умолчанию, поэтому `this` вместо этого присваивается `undefined`:

```
function foo() {
    "use strict";

    console.log( this.a );
}

var a = 2;

foo(); // TypeError: `this` is `undefined`
```

Тонкая, но важная подробность: хотя в целом эти правила связывания полностью зависят от места вызова, глобальный объект доступен для связывания по умолчанию только в том случае, если содержимое `foo()` не выполняется в режиме `strict`; действие режима `strict` для места вызова `foo()` роли не играет:

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

var a = 2;

(function(){
    "use strict";

    foo(); // 2
})();
```

Как правило, намеренно смешивать в коде части с включенным и отключенным режимом `strict` не рекомендуется. Вероятно, вся ваша программа должна работать либо в режиме `strict`, либо без него. Однако иногда в код приходится включать сторонние библиотеки, отличающиеся по состоянию режима strict от вашего кода, поэтому вы должны внимательно следить за этими тонкими подробностями совместимости.

Неявное связывание

Другое правило, которое необходимо учитывать, — наличие у места вызова контекстного объекта, также называемого *владельцем* (owner) или *содержащим объектом* (хотя эти альтернативные термины могут создавать путаницу).

Пример:

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

var obj = {
    a: 2,
    foo: foo
};

obj.foo(); // 2
```

Сначала обратите внимание на то, как `foo()` объявляется, а затем добавляется как ссылочное свойство в `obj`. Независимо от того, объявляется ли `foo()` для `obj` изначально или же добавляется как ссылка позднее (как в приведенном фрагменте), ни в одном из случаев объект `obj` не является «владельцем» функции и не «содержит» ее.

Тем не менее место вызова использует контекст `obj` для создания ссылки на функцию, поэтому можно сказать, что объект `obj` является «владельцем» или «содержит» ссылку на функцию на момент ее вызова.

Какое бы название вы ни выбрали для этого паттерна, в точке вызова `foo()` имени функции будет предшествовать ссылка на объект `obj`. Когда при обращении к функции указывается контекстный объект, правило неявного связывания требует, чтобы этот объект использовался для связывания `this` данного вызова функции. Поскольку `obj` становится `this` для вызова `foo()`, синтаксис `this.a` является синонимом `obj.a`.

Для места вызова важен только верхний/последний уровень цепочки ссылок на свойства объекта. Пример:

```
function foo() {
    console.log( this.a );
```

```
}

var obj2 = {
    a: 42,
    foo: foo
};

var obj1 = {
    a: 2,
    obj2: obj2
};

obj1.obj2.foo(); // 42
```

Неявная потеря this

Одна из самых распространенных проблем, возникающих при связывании, — когда функция с *неявным связыванием* теряет это связывание, что обычно означает возврат к связыванию по умолчанию — глобальному объекту или `undefined` в зависимости от действия режима `strict`.

Пример:

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

var obj = {
    a: 2,
    foo: foo
};

var bar = obj.foo; // Ссылка на функцию/синоним!
var a = "oops, global"; // `a` также является свойством
                        // глобального объекта

bar(); // "oops, global"
```

И хотя bar кажется ссылкой на obj.foo, в действительности это всего лишь еще одна ссылка на саму функцию foo. Кроме того, важно только место вызова, а здесь оно имеет вид bar() — простой вызов без префикса, а следовательно, здесь применяется связывание по умолчанию.

Более тонкий, более распространенный и более неожиданный пример встречается при передаче функции обратного вызова:

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

function doFoo(fn) {
    // `fn` - просто еще одна ссылка на `foo`

    fn(); // <-- место вызова!
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` также является свойством
                        // глобального объекта

doFoo( obj.foo ); // "oops, global"
```

Передача параметров — всего лишь неявное присваивание, а поскольку мы передаем функцию, происходит неявное присваивание ссылки, поэтому конечный результат будет таким же, как в предыдущем фрагменте.

А если передаваемая функция не написана вами, а встроена в язык? Ничего не меняется, результат будет тем же:

```
function foo() {
    console.log( this.a );
```

```
}

var obj = {
    a: 2,
    foo: foo
};

var a = "oops, global"; // `a` также является свойством
                        // глобального объекта

setTimeout( obj.foo, 100 ); // "oops, global"
```

Присмотритесь к следующей примитивной теоретической псевдореализации `setTimeout()`, предоставляемой во встроенном виде средой JavaScript:

```
function setTimeout(fn,delay) {
    // Сделать паузу продолжительностью `delay` миллисекунд
    fn(); // <-- место вызова!
}
```

Потеря связывания `this` обратными вызовами — явление вполне распространенное, как вы уже видели. Но возможны и другие сюрпризы, когда функция, которой передается обратный вызов, намеренно изменяет `this` для вызова. Обработчики событий в популярных JavaScript нередко заставляют обратные вызовы иметь значение `this`, указывающее, например, на элемент DOM, инициировавший событие. Хотя иногда это бывает полезно, в других случаях такое поведение откровенно бесит. К сожалению, эти средства редко предоставляют вам выбор.

В любом случае `this` изменяется неожиданно, а вы не управляете тем, как будет выполняться ваша ссылка на функцию обратного вызова, поэтому вы (пока) не можете управлять местом вызова для обеспечения желательного связывания. Вскоре вы увидите, как решить эту проблему посредством «настройки» `this`.

Явное связывание

При неявном связывании, как вы только что видели, необходимо изменить объект для включения ссылки на функцию и использовать эту ссылку на функцию в свойстве для неявного (косвенного) связывания `this` с объектом.

Но что, если вы хотите заставить вызов функции использовать конкретный объект для связывания `this`, без включения свойства со ссылкой на функцию в объект?

У всех функций в языке имеются средства (`[[Prototype]]` — об этом см. далее), которые могут пригодиться для решения этой задачи. А именно — функции содержат методы `call(..)` и `apply(..)`. Строго говоря, управляющие среды JavaScript иногда предоставляют достаточно специализированные (мягко говоря) функции, не обладающие такой функциональностью. Тем не менее такие случаи редки. Для подавляющего большинства готовых функций — и конечно, всех функций, написанных вами, — доступны вызовы `call(..)` и `apply(..)`.

Как работают эти средства? Они получают в первом параметре объект, который должен использоваться для `this`, а затем вызывают функцию с заданным значением `this`. Так как вы напрямую указываете, какое значение должно быть присвоено `this`, мы будем называть этот способ связывания *явным*.

Пример:

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

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Вызов foo с явным связыванием foo.call(..) позволяет принудительно задать его this значение obj.

Если передать простое примитивное значение (типа string, boolean или number) для связывания this, это примитивное значение будет преобразовано в объектную форму (new String(..), new Boolean(..) или new Number(..) соответственно). Часто это преобразование называется *упаковкой* (boxing).

В отношении связывания this поведение call(..) и apply(..) идентично. Они отличаются по дополнительным параметрам, но в настоящий момент этот вопрос для нас интереса не представляет.

К сожалению, явное связывание также не решает проблему, упоминавшуюся ранее — «потерю» функцией ее предполагаемого связывания this, замену его фреймворками и т. д.

Жесткое связывание

Впрочем, существует паттерн на базе явного связывания, который решает эту проблему.

Пример:

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

var obj = {
    a: 2
};

var bar = function() {
    foo.call( obj );
```

```
};

bar(); // 2
setTimeout( bar, 100 ); // 2

// У жестко связанной функции `bar` значение `this` не может
// заменяться
bar.call( window ); // 2
```

А теперь разберемся, как работает это решение. Мы создаем функцию `bar()`, которая во внутренней реализации вручную вызывает `foo.call(obj)`, принудительно вызывая `foo` со связыванием `obj` для `this`. Неважно, как позднее будет вызываться функция `bar` — она всегда вручную будет вызывать `foo` с `obj`. Такой способ связывания явно выражен и надежен; мы будем называть его *жестким связыванием*.

Самый типичный способ упаковки функции с жестким связыванием создает сквозную передачу любых переданных аргументов и полученного возвращаемого значения:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = function() {
    return foo.apply( obj, arguments );
};

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Другой способ выражения этого паттерна основан на создании вспомогательной функции, пригодной для повторного использования:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

// Простая вспомогательная функция `bind`
function bind(fn, obj) {
    return function() {
        return fn.apply( obj, arguments );
    };
}

var obj = {
    a: 2
};

var bar = bind( foo, obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

Так как жесткое связывание является чрезвычайно распространенным паттерном, в ES5 для него была определена встроенная реализация `Function.prototype.bind`, которая используется следующим образом:

```
function foo(something) {
    console.log( this.a, something );
    return this.a + something;
}

var obj = {
    a: 2
};

var bar = foo.bind( obj );

var b = bar( 3 ); // 2 3
console.log( b ); // 5
```

`bind(..)` возвращает новую функцию, жестко запрограмированную для вызова исходной функции с заданным вами контекстом `this`.

«Контексты» вызовов API

Функции многих библиотек, а также многие новые встроенные функции языка JavaScript и управляющей среды поддерживают необязательный параметр (обычно с именем `context`), который избавляет вас от необходимости использовать `bind()`, чтобы ваша функция обратного вызова гарантированно использовала конкретное значение `this`.

Пример:

```
function foo(el) {
    console.log( el, this.id );
}

var obj = {
    id: "awesome"
};

// Использовать `obj` как `this` для вызовов `foo(..)`
[1, 2, 3].forEach( foo, obj );
// 1 awesome  2 awesome  3 awesome
```

Во внутренней реализации эти функции почти наверняка используют явное связывание вызовом `call(..)` или `apply(..)`, избавляя вас от лишних хлопот.

Связывание new

Четвертое и последнее правило связывания `this` заставляет нас пересмотреть очень распространенное заблуждение, касающееся функций и объектов в JavaScript.

В традиционных языках, основанных на использовании классов, «конструкторы» представляют собой специальные методы, присоединенные к классам. При создании экземпляра класса оператором `new` вызывается конструктор этого класса. Обычно это выглядит примерно так:

```
something = new MyClass(..);
```

В JavaScript существует оператор `new`, а паттерн его использования почти не отличается от того, что мы видим в языках с использованием классов; большинство разработчиков предполагает, что механизм JavaScript делает нечто похожее. Однако на самом деле использование `new` в JS не имеет отношения к функциональности, относящейся к классам.

Сначала заново определим понятие «конструктора» в JavaScript. В JS конструкторы представляют собой обычные функции, которые вызываются с указанием оператора `new` перед ними. Они не присоединяются к классам и не создают их экземпляры. Они даже не являются особой разновидностью функций. Это самые обычные функции, которые наделяются новым смыслом из-за использования `new` при их вызове.

Для примера возьмем функцию `Number(..)`, которая работает как конструктор. Цитата из спецификации ES5.1:

> 15.7.2. Конструктор Number
>
> Когда функция `Number` вызывается как часть нового выражения, она является конструктором: эта функция инициализирует созданный объект.

Таким образом, практически любая функция, включая функции встроенных объектов (такие, как `Number(..)`), может быть вызвана после `new`, и тогда этот вызов функции становится вызовом кон-

структора. Здесь проявляется важное, но неочевидное различие: такого понятия, как «функция-конструктор», не существует. Существуют вызовы — конструкторы функций.

Когда функция вызывается после оператора `new` (такие вызовы называются *вызовами-конструкторами*), автоматически выполняются следующие действия:

1. Создается (конструируется) новый объект.
2. Производится связывание сконструированного объекта с `[[Prototype]]`.
3. Сконструированный объект назначается в качестве связывания `this` для этого вызова функции.
4. Если функция не возвращает свой альтернативный объект, вызов функции *автоматически* возвращает сконструированный объект.

Шаги 1, 3 и 4 относятся к нашему текущему обсуждению. Шаг 2 мы пока пропустим и вернемся к нему в главе 10.

Рассмотрим следующий код:

```
function foo(a) {
    this.a = a;
}

var bar = new foo( 2 );
console.log( bar.a ); // 2
```

Вызывая `foo(..)` после `new`, мы конструируем новый объект и назначаем его в качестве значения `this` для вызова `foo(..)`. Итак, `new` — последний способ связывания `this` для вызова функции. Назовем его *связыванием new*.

Все по порядку

Итак, мы открыли четыре правила связывания `this` при вызовах функций. Остается совсем немного: найти место вызова и проанализировать его, чтобы узнать, какое из правил применяется в данном случае. Но что, если место вызова подходит для нескольких правил? Правила должны обладать приоритетом, поэтому выясним, в каком порядке они должны применяться.

Очевидно, правило связывания по умолчанию имеет самый низкий приоритет из четырех. Это правило можно отложить.

Что обладает более высоким приоритетом: *неявное связывание* или *явное*? Проверим:

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

var obj1 = {
    a: 2,
    foo: foo
};

var obj2 = {
    a: 3,
    foo: foo
};

obj1.foo(); // 2
obj2.foo(); // 3

obj1.foo.call( obj2 ); // 3
obj2.foo.call( obj1 ); // 2
```

Итак, явное связывание приоритетнее неявного, а следовательно, перед проверкой неявного связывания следует сначала проверить, действует ли явное связывание.

Теперь необходимо определить, какое место в схеме приоритетов занимает связывание new:

```
function foo(something) {
    this.a = something;
}

var obj1 = {
    foo: foo
};

var obj2 = {};

obj1.foo( 2 );
console.log( obj1.a ); // 2

obj1.foo.call( obj2, 3 );
console.log( obj2.a ); // 3

var bar = new obj1.foo( 4 );
console.log( obj1.a ); // 2
console.log( bar.a ); // 4
```

Понятно, связывание new обладает более высоким приоритетом, чем неявное связывание. Но как вы думаете, выше или ниже его приоритет, чем приоритет явного связывания?

`new` и `call/apply` не могут использоваться вместе, так что для прямого сравнения приоритета связывания new с приоритетом явного связывания невозможно использовать `new foo.call(obj1)`. Тем не менее для сравнения приоритетов двух правил можно воспользоваться жестким связыванием.

Прежде чем выяснять этот вопрос в коде, вспомните, как физически работает жесткое связывание — вызов `Function.prototype.bind(..)` создает новую функцию-обертку, которая жестко запрограммирована на то, чтобы игнорировать собственное связывание

this (каким бы оно ни было) и использовать предоставленное значение.

Казалось бы, очевидно предположить, что жесткое связывание (которое является формой явного связывания) обладает более высоким приоритетом, чем связывание new, а следовательно, не может замещаться связыванием new.

Проверим:

```
function foo(something) {
    this.a = something;
}

var obj1 = {};

var bar = foo.bind( obj1 );
bar( 2 );
console.log( obj1.a ); // 2

var baz = new bar( 3 );
console.log( obj1.a ); // 2
console.log( baz.a ); // 3
```

Вот это да! bar жестко связывается с obj1, но new bar(3) не меняет значение obj1.a на 3, как можно было ожидать. Вместо этого жестко связанный (с obj1) вызов bar(..) был переопределен с new. Так как вызов new был применен, мы получили обратно новый созданный объект, которому было присвоено имя baz, и мы видим, что baz.a действительно имеет значение 3.

Это выглядит довольно неожиданно, если вернуться к «условной» вспомогательной функции bind:

```
function bind(fn, obj) {
    return function() {
        fn.apply( obj, arguments );
    };
}
```

Как бы вы ни анализировали код вспомогательной функции, вызов оператора `new` никак не может переопределить жесткое связывание с `obj`, как мы только что видели.

Однако встроенная функция `Function.prototype.bind(..)` в ES5 устроена намного сложнее. Ниже приведен (слегка переформатированный) полифил, предоставляемый на странице MDN для `bind(..)`:

```
if (!Function.prototype.bind) {
    Function.prototype.bind = function(oThis) {
        if (typeof this !== "function") {
            // Ближайший возможный аналог внутренней функции
            // IsCallable из ECMAScript 5
            throw new TypeError(
               "Function.prototype.bind - what is trying " +
               "to be bound is not callable"
            );
        }

        var aArgs = Array.prototype.slice.call( arguments, 1 ),
            fToBind = this,
            fNOP = function(){},
            fBound = function(){
                return fToBind.apply(
                    (
                        this instanceof fNOP &&
                        oThis ? this : oThis
                    ),
                    aArgs.concat(
                       Array.prototype.slice.call( arguments )
                    );
            }
        ;

        fNOP.prototype = this.prototype;
        fBound.prototype = new fNOP();

        return fBound;
    };
}
```

Полифил `bind(..)`, приведенный выше, отличается от встроенной версии `bind(..)` из ES5 в отношении функций с жестким связыванием, которые будут использоваться с `new` (о том, для чего это может быть полезно, рассказано ниже). Так как полифил не может создать функцию без `.prototype`, как это делает встроенная функция, то используются нетривиальные обходные пути для аппроксимации того же поведения. Будьте осторожны, если собираетесь использовать `new` с функцией с жестким связыванием и работа вашей программы зависит от этого поведения.

Часть, которая делает возможным переопределение с `new`:

```
this instanceof fNOP &&
oThis ? this : oThis

// ... и:

fNOP.prototype = this.prototype;
fBound.prototype = new fNOP();
```

Мы не будем подробно объяснять, как работает этот трюк (это сложно, к тому же тема выходит за рамки книги), но по сути функция определяет, была ли функция с жестким связыванием вызвана с `new` (в результате чего в качестве `this` будет использоваться новый сконструированный объект), и если была — использует новый созданный `this` вместо ранее заданной жесткой привязки.

Почему возможность переопределения жесткого связывания с `new` может быть полезной?

Прежде всего это делается для создания функции (которая может использоваться с `new` для построения объектов), которая фактически игнорирует жесткую привязку `this`, но предварительно задает некоторые (или все) аргументы функции. Одна из особен-

ностей `bind(..)` заключается в том, что любые аргументы, переданные после первого аргумента `this`, по умолчанию становятся стандартными аргументами нижележащей функции (в технической терминологии это называется «частичным применением», что является подмножеством «каррирования» (currying)). Пример:

```
function foo(p1,p2) {
        this.val = p1 + p2;
}

// Здесь используется `null`, потому что в данном случае
// жесткая привязка `this` нас не интересует,
// к тому же она все равно будет переопределена
// вызовом `new`!
var bar = foo.bind( null, "p1" );

var baz = new bar( "p2" );

baz.val; // p1p2
```

Определение this

А теперь можно собрать воедино правила определения `this` по месту вызова функции в порядке их приоритетов. Задайте себе следующие вопросы в указанном порядке и остановитесь, когда найдете первое подходящее правило.

1. Функция вызвана с `new` (*связывание new*)? Если да, то `this` содержит новый сконструированный объект.

   ```
   var bar = new foo()
   ```

2. Функция вызвана с `call` или `apply` (явное связывание), даже скрытыми в жесткой привязке `bind`? Если да, то `this` содержит явно заданный объект.

   ```
   var bar = foo.call( obj2 )
   ```

3. Функция вызвана с контекстом (неявное связывание), также называемым объектом-владельцем или содержащим объектом? Если да, то `this` содержит контекстный объект.

   ```
   var bar = obj1.foo()
   ```

4. В остальных случаях используется `this` по умолчанию (привязка по умолчанию). Если действует режим `strict`, выбирается `undefined`, а если нет — глобальный объект:

   ```
   var bar = foo()
   ```

Вот и все. Вот и все, что необходимо для понимания правил связывания `this` для нормальных вызовов функций. Ну... почти все.

Исключения связывания

Как обычно, у «правил» находятся исключения.

Поведение связывания `this` в некоторых ситуациях может быть довольно странным: вы рассчитываете на другое связывание, а получаете поведение правила связывания по умолчанию.

Игнорирование this

Если передать `call`, `apply` или `bind` в параметре связывания `this` значение `null` или `undefined`, эти значения фактически игнорируются, и к вызову применяется правило *связывания по умолчанию*:

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

var a = 2;

foo.call( null ); // 2
```

Зачем намеренно передавать `null` для связывания `this`?

`apply(..)` довольно часто применяется для распределения массивов значений по параметрам вызова функций. Также `bind()` может выполнять каррирование параметров, что может быть очень полезно:

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// Распределение массива по параметрам
foo.apply( null, [2, 3] ); // a:2, b:3

// Каррирование вызовом `bind(..)`
var bar = foo.bind( null, 2 );
bar( 3 ); // a:2, b:3
```

Обе функции требуют передачи связывания `this` в первом параметре. Если сами функции не обращают внимания на `this`, понадобится значение-заполнитель; как показывает этот фрагмент, `null` может показаться разумным кандидатом.

В книге эта возможность не рассматривается, но в ES6 появился оператор распределения `...`, позволяющий на синтаксическом уровне «распределить» массив по параметрам без использования `apply(..)`, например `foo(...[1,2])`, что эквивалентно `foo(1,2)`; таким образом, лишнее связывание предотвращается на уровне синтаксиса. К сожалению, в ES6 нет синтаксической замены для каррирования, так что параметр `this` вызова `bind(..)` все равно приходится учитывать.

Тем не менее, если вы начнете всегда использовать `null` там, где вас не интересует связывание `this`, возникает небольшая скрытая «опасность». Если вы когда-либо используете это значение при

вызове функции (например, функции сторонней библиотеки, которая вам неподконтрольна) и эта функция использует ссылку `this`, правило связывания по умолчанию означает, что она может случайно обратиться к глобальному объекту (`window` в браузере) — или, хуже того, изменить его!

Очевидно, подобные ловушки могут создавать разнообразные ошибки, которые очень трудно диагностировать и выявить.

Безопасность при использовании this

Возможна чуть более «безопасная» практика, передача для `this` специально созданного объекта, который гарантированно не создаст неприятных побочных эффектов в вашей программе. Заимствуя терминологию из сетей (и военного дела), мы создаем *DMZ*-объект (Demilitarized Zone, то есть «демилитаризированная зона»), то есть совершенно пустой объект без делегирования.

Если всегда передавать DMZ-объект для игнорирования связывания `this`, которое, скорее всего, не будет представлять для нас интереса, можно быть уверенными в том, что все скрытое/непредвиденное использование `this` будет ограничено пустым объектом; таким образом, глобальный объект программы будет изолирован от побочных эффектов.

Так как этот объект совершенно пуст, я предпочитаю присваивать ему имя переменной `ø` (математический знак пустого множества в нижнем регистре). На многих клавиатурах (скажем, в US-раскладке на Mac) этот знак легко вводится комбинацией `⌥+o` (`Option+o`). Некоторые системы также позволяют назначить свои комбинации клавиш для конкретных символов. Конечно, если знак `ø` вам не нравится или на вашей клавиатуре его вводить слишком трудно, вы можете присвоить ему любое имя на свое усмотрение.

Проще всего создать совершенно пустой объект вызовом `Object.create(null)` (см. главу 10). Вызов `Object.create(null)` аналогичен

{ }, но без делегирования Object.prototype, так что объект получается «более пустым», чем просто { }:

```
function foo(a,b) {
    console.log( "a:" + a + ", b:" + b );
}

// Пустой DMZ-объект
var ø = Object.create( null );

// распределение массива по параметрам
foo.apply( ø, [2, 3] ); // a:2, b:3

// каррирование вызовом `bind(..)`
var bar = foo.bind( ø, 2 );
bar( 3 ); // a:2, b:3
```

Такое решение не только функционально «безопаснее», оно с точки зрения синтаксиса лучше ø, потому что на семантическом уровне передает намерение «Я хочу, чтобы значение this было пустым» чуть более очевидно, чем null. Тем не менее вы можете присвоить DMZ-объекту то имя, которое считаете нужным.

Косвенные ссылки

Также следует помнить еще об одном обстоятельстве: вы можете (намеренно или нет) создавать «косвенные ссылки» на функции. В этих случаях при вызове ссылки на функцию также применяется правило связывания по умолчанию. Один из самых распространенных вариантов появления косвенных ссылок в программах встречается при присваивании:

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

var a = 2;
```

```
var o = { a: 3, foo: foo };
var p = { a: 4 };

o.foo(); // 3
(p.foo = o.foo)(); // 2
```

Результирующим значением выражения присваивания `p.foo = o.foo` является ссылка на нижележащий объект функции. Отсюда следует, что фактическим местом вызова будет просто `foo()`, а не `p.foo()` или `o.foo()`, как можно было ожидать. Согласно системе правил, упомянутых ранее, применяется правило *связывания по умолчанию*.

Напомню: независимо от того, как вы приходите к вызову функции, использующей правило связывания по умолчанию, связываемое значение по умолчанию определяется действующим режимом `strict` для содержимого вызванной функции: глобальный объект (режим `strict` не действует) или `undefined` (режим `strict`).

Мягкое связывание

Ранее было показано, что жесткое связывание — одна из стратегий, предотвращающих случайный возврат вызова функции к правилу связывания по умолчанию, для чего функция принудительно связывается с конкретным значением `this` (если только не использовать `new` для переопределения). Проблема в том, что жесткое связывание сильно сокращает гибкость функции, предотвращая ручное переопределение `this` посредством неявного связывания или даже попыток явного связывания.

Было бы удобно иметь возможность передачи другого значения для связывания по умолчанию (не глобального объекта и не `undefined`), чтобы при этом можно было выполнить ручное связывание `this` методами неявного или явного связывания.

Мы можем создать функцию так называемого *мягкого связывания*, эмулирующую желаемое поведение:

```
if (!Function.prototype.softBind) {
    Function.prototype.softBind = function(obj) {
        var fn = this,
            curried = [].slice.call( arguments, 1 ),
            bound = function bound() {
                return fn.apply(
                    (!this ||
                        (typeof window !== "undefined" &&
                            this === window) ||
                        (typeof global !== "undefined" &&
                            this === global)
                    ) ? obj : this,
                    curried.concat.apply( curried, arguments )
                );
            };
        bound.prototype = Object.create( fn.prototype );
        return bound;
    };
}
```

Приведенная функция `softBind(..)` работает аналогично встроенной функции ES5 `bind(..)`, не считая поведения мягкого связывания. Заданная функция упаковывается в логику, которая проверяет `this` во время вызова, и если это глобальный объект или `undefined` — использует заранее заданное альтернативное значение по умолчанию (`obj`). В остальных случаях `this` остается без изменений. Она также предоставляет необязательную возможность каррирования (см. выше обсуждение `bind(..)`).

Пример ее использования:

```
function foo() {
   console.log("name: " + this.name);
}

var obj = { name: "obj" },
    obj2 = { name: "obj2" },
```

```
    obj3 = { name: "obj3" };

var fooOBJ = foo.softBind( obj );

fooOBJ(); // name: obj

obj2.foo = foo.softBind(obj);
obj2.foo(); // name: obj2   <---- смотрите!!!

fooOBJ.call( obj3 ); // name: obj3   <---- смотрите!

setTimeout( obj2.foo, 10 );
// name: obj   <---- возврат к мягкому связыванию
```

Версия функции `foo()` с мягким связыванием может быть вручную связана по `this` с `obj2` или `obj3`, как показано, но она возвращается к `obj`, если должно применяться связывание по умолчанию.

Лексическое поведение this

Нормальные функции подчиняются четырем описанным правилам. Но в ES6 появилась особая разновидность функций, которая не использует эти правила: *стрелочные функции*.

Стрелочные функции обозначаются не ключевым словом `function`, а так называемым оператором «толстой стрелки» `=>`. Вместо четырех стандартных правил `this` стрелочные функции принимают связывание `this` от внешней области видимости (функциональной или глобальной).

А теперь продемонстрируем лексическую область видимости стрелочной функции:

```
function foo() {
    // Вернуть стрелочную функцию
    return (a) => {
        // `this` здесь лексически наследуется от `foo()`
        console.log( this.a );
```

```
    };
}

var obj1 = {
    a: 2
};

var obj2 = {
    a: 3
};

var bar = foo.call( obj1 );
bar.call( obj2 ); // 2, не 3!
```

Стрелочная функция, созданная в `foo()`, лексически захватывает значение `this` функции `foo()` на момент вызова. Так как `foo()` была связана по `this` с `obj1`, `bar` (ссылка на возвращенную стрелочную функцию) также будет связана по `this` с `obj1`. Лексическое связывание стрелочной функции не может быть переопределено (даже с `new`).

Вероятно, самый распространенный сценарий использования — это обратные вызовы (например, при использовании обработчиков событий или таймеров):

```
function foo() {
    setTimeout(() => {
        // `this` здесь лексически наследуется от `foo()`
        console.log( this.a );
    },100);
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Хотя стрелочные функции предоставляют альтернативу для использования `bind(..)` с функциями для обеспечения нужного

значения this, что кажется привлекательным, важно заметить, что по сути они блокируют традиционный механизм this в пользу более понятной лексической области видимости. До появления ES6 уже существовал довольно распространенный паттерн, который по духу был почти неотличим от стрелочных функций ES6:

```
function foo() {
    var self = this; // лексический захват `this`
    setTimeout( function(){
        console.log( self.a );
    }, 100 );
}

var obj = {
    a: 2
};

foo.call( obj ); // 2
```

Хотя и self = this, и стрелочные функции могут показаться хорошими «решениями» для тех, кто не хочет использовать bind(..), на самом деле это попытки сбежать от конструкции this, вместо того чтобы попытаться понять и принять ее.

Если вы пишете код в стиле this, но в большинстве случаев подавляете этот механизм лексическим self = this или «трюками» со стрелочными функциями, возможно, вам стоит выбрать один из следующих вариантов:

1. Использовать только лексическую область видимости и забыть о пустых претензиях на код «в стиле this».

2. Полностью принять механизмы this, включая использование bind() там, где это необходимо, и по возможности избегать self = this и трюков со стрелочными функциями.

В программах можно эффективно использовать оба стиля кода (лексический и this), однако смешение двух механизмов внутри

одной функции (и даже для одинаковых видов поиска) обычно усложняет сопровождение кода и создает ощущение, что автор кода уж слишком старается блеснуть умом.

Итоги

Чтобы определить связывание `this` для выполняемой функции, необходимо найти непосредственное место вызова этой функции. После его нахождения к месту вызова применяются четыре правила (в указанном порядке приоритетов):

1. Вызывается с `new`? Используется только что сконструированный объект.
2. Вызывается с `call` или `apply` (или `bind`)? Используется заданный объект.
3. Вызывается с контекстным объектом, которому принадлежит данный вызов? Используется этот контекстный объект.
4. По умолчанию: `undefined` в режиме `strict`, глобальный объект в противном случае.

Будьте осторожны со случайной/непреднамеренной активизацией правила связывания по умолчанию. В тех случаях, когда вы хотите «безопасно» игнорировать связывание `this`, стоит использовать DMZ-объект (например, `ø = Object.create(null)`), защищающий глобальный объект от непредвиденных побочных эффектов.

Вместо четырех стандартных правил связывания стрелочные функции ES6 используют для связывания `this` лексическую область видимости — иначе говоря, они наследуют значение `this` (каким бы оно ни было) от внешнего вызова функции. Фактически они служат синтаксической заменой `self = this` в коде до появления ES6.

8 Объекты

В главах 6 и 7 вы узнали, как связывание `this` может указывать на разные объекты в зависимости от места вызова функции. Но что такое объекты и почему на них нужно указывать? В этой главе тема объектов будет рассмотрена более подробно.

Синтаксис

Объекты существуют в двух формах: *декларативной* (литеральной) и *сконструированной*.

Литеральный синтаксис объекта выглядит так:

```
var myObj = {
    key: value
    // ...
};
```

Сконструированная форма выглядит так:

```
var myObj = new Object();
myObj.key = value;
```

Объекты, созданные в сконструированной и литеральной форме, идентичны. Единственное различие заключается в том, что в литеральном объявлении можно добавить одну или несколько пар «ключ/значение», тогда как в сконструированной форме свойства должны добавляться по одному.

Использование «сконструированной формы» для создания объектов в высшей степени нетипично. Вам почти всегда стоит использовать синтаксис литеральной формы. Это относится и к большинству встроенных объектов (об этом ниже).

Тип

Объекты — общие структурные элементы, на которых строится большая часть JS. В JS существуют шесть основных типов (в спецификации они называются «языковыми типами»):

- `string`
- `number`
- `boolean`
- `null`
- `undefined`
- `object`

Учтите, что *простые примитивы* (`string`, `boolean`, `number`, `null` и `undefined`) сами по себе объектами не являются. `null` иногда причисляют к объектному типу, но это недоразумение происходит от ошибки в языке, из-за которой `typeof null` неправильно возвращает строку `"object"`. В действительности `null` является отдельным примитивным типом.

Существует распространенное заблуждение, будто «в JavaScript нет ничего, кроме объектов». Очевидно, это не так.

При этом существует несколько специальных объектных подтипов, которые можно назвать *сложными примитивами*.

`function` является подтипом `object` (формально — «вызываемый объект»). Функции JS называются «первоклассными» в том отношении, что они по сути являются обычными объектами (с добавленной семантикой поведения вызова), поэтому с ними можно работать как с любыми другими обычными объектами.

Массивы тоже являются разновидностью объектов с расширенным поведением. Содержимое массива чуть более структурировано, чем у обобщенного объекта.

Встроенные объекты

Также существует несколько других подтипов, обычно называемых *встроенными объектами*. Имена некоторых из них вроде бы указывают на их прямую связь с простыми примитивными аналогами, но как вы вскоре увидите, на самом деле связь между ними более сложна.

- `String`
- `Number`
- `Boolean`
- `Object`
- `Function`
- `Array`
- `Date`

- `RegExp`
- `Error`

Встроенные объекты выглядят как обычные типы и даже классы, если опираться на их сходство с другими языками (например, классом `String` в языке Java).

Но в JS они в действительности представляют собой встроенные функции. Каждая из встроенных функций может использоваться как конструктор (то есть вызов функции с оператором `new`), результатом вызова которого является создание объекта заданного подтипа. Пример:

```
var strPrimitive = "I am a string";
typeof strPrimitive; // "string"
strPrimitive instanceof String; // false
var strObject = new String( "I am a string" );
typeof strObject; // "object"
strObject instanceof String; // true
// проверка подтипа объекта
Object.prototype.toString.call( strObject ); // [object String]
```

В одной из последующих глав будет подробно показано, как работает `Object.prototype.toString...`, но пока мы можем проанализировать внутренний подтип при помощи базового метода `toString()` по умолчанию. Как видите, `strObject` является объектом, который в действительности был создан конструктором `String`.

Примитивное значение `"I am a string"` объектом *не является* — это примитивный литерал и неизменяемое значение. Для выполнения с ним операций (например, проверки длины, обращения к отдельным символам содержимого и т. д.) необходим объект `String`.

К счастью, язык автоматически преобразует строковый примитив в объект `String` тогда, когда это необходимо; а это означает, что

вам почти никогда не понадобится явно создавать форму `Object`. В сообществе JS считается, что для значений следует по возможности использовать литеральную форму вместо сконструированной формы объекта.

Пример:

```
var strPrimitive = "I am a string";

console.log( strPrimitive.length ); // 13

console.log( strPrimitive.charAt( 3 ) ); // "m"
```

В обоих случаях мы вызываем свойство или метод для строкового примитива, а движок автоматически преобразует его в объект `String`, чтобы обращение к свойству/методу работало.

Аналогичное преобразование выполняется между числовым литеральным примитивом `42` и объектной оберткой `new Number(42)` при использовании таких методов, как `42.359.toFixed(2)`. То же самое происходит при получении объектов `Boolean` из примитивов `"boolean"`.

У `null` и `undefined` не существует объектных оберток, только примитивные значения. С другой стороны, значения `Date` могут создаваться только в сконструированной форме, у них нет аналогичной литеральной формы.

Объекты `Object`, массивы `Array`, функции `Function` и регулярные выражения `RegExp` — все они являются объектами независимо от того, используется ли для них литеральная или сконструированная форма. В некоторых ситуациях сконструированная форма предоставляет большую гибкость при создании объекта, чем аналогичная литеральная форма. Так как объекты могут создаваться любым из двух способов, более простая литеральная форма почти повсеместно считается предпочтительной. Сконструи-

рованная форма используется только в тех случаях, когда вам необходимы расширенные возможности.

Объекты `Error` редко создаются в коде явно; обычно они создаются автоматически при выдаче исключений. Они могут создаваться в сконструированной форме `new Error(..)`, но такая необходимость возникает редко.

Содержимое

Как упоминалось ранее, содержимое объекта состоит из значений (любого типа), хранимых в *свойствах* — областях памяти, которым присвоены имена. Важный момент: термин «содержимое» вроде бы подразумевает, что значения хранятся *внутри* объекта, но это лишь иллюзия. Способ хранения значений зависит от реализации, и данные вполне могут храниться за пределами объекта-контейнера. В контейнере хранятся имена свойств, которые работают как указатели (а точнее, *ссылки*) на место хранения значений.

Пример:

```
var myObject = {
    a: 2
};

myObject.a; // 2

myObject["a"]; // 2
```

Чтобы обратиться к значению в *области памяти* `a` объекта `myObject`, необходимо использовать оператор `.` или оператор `[ ]`. Синтаксис `.a` обычно называется «обращением к свойству» (или «точечной формой записи»), тогда как синтаксис `["a"]` чаще называется «обращением к ключу» (или «скобочной формой записи»). На самом деле обе формы обращаются к одной области

памяти и извлекают одно значение 2, так что эти формы могут использоваться как взаимозаменяемые. С этого момента и далее в книге будет использоваться более общий термин «обращение к свойству».

Главное различие между двумя вариантами синтаксиса заключается в том, что после оператора `.` должно указываться `Identifier`-совместимое,тогда как в синтаксисе `[".."]` в качестве имени свойства может использоваться практически любая UTF-8/Юникод-совместимая строка. Например, для обращения к свойству с именем `"Super-Fun!"` необходимо использовать синтаксис `["Super-Fun!"]`, поскольку `Super-Fun!` не является действительным именем свойства `Identifier`.

Кроме того, поскольку синтаксис `[".."]` использует значение строки для определения места хранения данных, это означает, что такое значение может строиться программными средствами:

```
var myObject = {
    a: 2
};

var idx;
if (wantA) {
    idx = "a";
}

// ...

console.log( myObject[idx] ); // 2
```

В объектах имена свойств *всегда* являются строками. Если в качестве свойства используется любое другое значение, кроме `string` (примитива), оно сначала будет преобразовано в строку. Это относится и к числам, которые обычно используются в качестве индексов массивов, так что будьте осторожны и не путайте использование чисел между объектами и массивами:

```
var myObject = { };

myObject[true] = "foo";
myObject[3] = "bar";
myObject[myObject] = "baz";

myObject["true"]; // "foo"
myObject["3"]; // "bar"
myObject["[object Object]"]; // "baz"
```

Вычисление имен свойств

Синтаксис обращения по ключу `myObject[..]`, описанный выше, удобен в тех случаях, когда в качестве ключа должно использоваться вычисляемое значение выражения, например `myObject[prefix + name]`. Впрочем, при объявлении объектов с использованием синтаксиса объектных литералов это особой пользы не принесет.

В ES6 появились *вычисляемые имена свойств*: в позиции «ключ-имя» объявления объектного литерала указывается выражение, заключенное в квадратные скобки:

```
var prefix = "foo";

var myObject = {
    [prefix + "bar"]: "hello",
    [prefix + "baz"]: "world"
};

myObject["foobar"]; // hello
myObject["foobaz"]; // world
```

Пожалуй, вычисляемые имена свойств чаще всего будут применяться при работе с символическими именами (`Symbol`) ES6, которые в этой книге подробно не рассматриваются. Вкратце это новый примитивный тип данных с непрозрачным значени-

ем, которое невозможно угадать (с технической точки зрения — значение `string`). Работать с фактическим значением `Symbol` (которое может теоретически различаться в разных движках JS) категорически не рекомендуется, поэтому вы будете использовать имя `Symbol`, например `Symbol.Something` (просто вымышленное имя):

```
var myObject = {
    [Symbol.Something]: "hello world"
};
```

Свойства и методы

Некоторые разработчики выделяют в особую категорию обращения к свойствам объекта, значения которых представляют собой функции. Возникает соблазнительная идея считать, что функция «принадлежит» объекту, а в других языках функции, принадлежащие объектам (или «классам»), называются «методами». Соответственно вместо «обращений к свойствам» часто приходится слышать об «обращениях к методам».

Интересно, что такое же различие обозначено в спецификации.

Формально функции никогда не «принадлежат» объектам, поэтому разговоры о том, что функция, к которой вы обращаетесь по ссылке на объект, автоматически становится методом, выглядят как определенная семантическая натяжка.

Правда, в некоторых функциях используется ссылка `this`, и *иногда* эти ссылки соответствуют объекту на месте вызова. Но такое использование не делает функцию «методом» в большей степени, чем любую другую функцию, так как ссылка `this` связывается динамически во время выполнения на месте вызова, а следовательно, ее привязку к объекту можно назвать в лучшем случае косвенной.

Каждый раз, когда вы обращаетесь к свойству объекта, происходит обращение к свойству независимо от типа значения, которое вы получаете. Если в результате обращения к свойству вы получаете функцию, она от этого не становится «методом» в результате какого-то волшебства. В функции, полученной при обращении к свойству, нет ничего особенного (если не считать неявного связывания `this`, о чем говорилось ранее).

Пример:

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

var someFoo = foo; // Ссылка на переменную `foo`

var myObject = {
    someFoo: foo
};

foo; // function foo(){..}

someFoo; // function foo(){..}

myObject.someFoo; // function foo(){..}
```

`someFoo` и `myObject.someFoo` — просто две разные ссылки на одну функцию, и ни одна из них не подразумевает, что в функции есть что-то особенное или она «принадлежит» любому другому объекту. Если бы функция `foo()` была определена так, что внутри нее использовалась бы ссылка `this`, то неявное связывание `myObject.someFoo` стало бы единственным заметным различием между двумя ссылками. Называть какую-либо из этих ссылок «методом» нелогично.

На это можно возразить, что функция становится методом не при определении, а во время выполнения только для конкретного вызова в зависимости от того, как именно она вызывается (с контек-

стом ссылки на объект или без нее — за подробностями обращайтесь к главе 7). А впрочем, даже эта интерпретация кажется немного преувеличенной.

Пожалуй, надежнее всего считать, что термины «функция» и «метод» являются синонимами в JavaScript.

В ES6 была добавлена ссылка super, обычно предназначенная для использования с классами (см. приложение Г). Поведение super (статическое связывание вместо позднего связывания, как с this) придает веса идее о том, что функция, связываемая через super в другом месте, скорее является «методом», нежели «функцией». Но и здесь речь скорее идет о нетривиальных семантических (и механических) нюансах.

Даже когда вы объявляете функциональное выражение как часть объектного литерала, это вовсе не означает, что функция *принадлежит* объекту в большей степени — все ссылки все равно указывают на один объект функции:

```
var myObject = {
    foo: function foo() {
        console.log( "foo" );
    }
};

var someFoo = myObject.foo;

someFoo; // function foo(){..}

myObject.foo; // function foo(){..}
```

В главе 11 будет рассмотрена сокращенная запись ES6 для foo из этого примера: синтаксис объявления `function foo() { .. }` в объектном литерале.

Массивы

Массивы также используют форму обращения `[ ]`, но как упоминалось ранее, способ и место хранения значений у них несколько сильнее структурированы (при отсутствии ограничений на *типы* хранимых значений). Массивы рассчитаны на числовое индексирование; это означает, что значения хранятся в позициях, обычно называемых *индексами*, которые принимают положительные целые значения (например, `0` и `42`):

```
var myArray = [ "foo", 42, "bar" ];

myArray.length; // 3

myArray[0]; // "foo"

myArray[2]; // "bar"
```

Массивы являются объектами. А значит, наряду с элементами, которые обозначаются положительными целыми числами, в массив также можно добавлять свойства:

```
var myArray = [ "foo", 42, "bar" ];

myArray.baz = "baz";

myArray.length; // 3

myArray.baz; // "baz"
```

Обратите внимание: добавление именованных свойств (независимо от синтаксиса операторов `.` или `[ ]`) не изменяет длину массива.

Массив можно использовать как простой объект «ключ/значение» без добавления числовых индексов, но это плохая мысль, потому что массивы обладают поведением и оптимизацией, ориентиро-

ванными на их специфику; то же относится к простым объектам. Используйте объекты для хранения пар «ключ/значение», а массивы — для хранения значений с числовыми индексами.

Будьте внимательны: если вы пытаетесь добавить свойство в массив, но имя свойства *выглядит* как число, оно вместо этого будет добавлено как числовой индекс (что приведет к изменению содержимого массива):

```
var myArray = [ "foo", 42, "bar" ];

myArray["3"] = "baz";

myArray.length; // 4

myArray[3]; // "baz"
```

Дублирование объектов

Один из самых частых вопросов, которые приходится слышать от начинающих JS-разработчиков, — как продублировать объект, то есть создать его копию? Казалось бы, должен существовать встроенный метод `copy()`, не правда ли? Оказывается, все немного сложнее — непонятно, какой алгоритм должен использоваться для дублирования объекта по умолчанию. Например, возьмем следующий объект:

```
function anotherFunction() { /*..*/ }

var anotherObject = {
    c: true
};

var anotherArray = [];

var myObject = {
```

```
    a: 2,
    b: anotherObject, // ссылка, а не копия!
    c: anotherArray, // другая ссылка!
    d: anotherFunction
};

anotherArray.push( anotherObject, myObject );
```

Что именно должно считаться представлением копии `myObject`?

Для начала нужно ответить, должна ли копия быть *поверхностной* или *глубокой*? У *поверхностной копии* значение `a` нового объекта будет копией значения 2, но свойства `b`, `c` и `d` будут всего лишь ссылками на те же области памяти, на которые указывают ссылки исходного объекта. *Глубокое копирование* создаст копии не только `myObject`, но и `anotherObject` и `anotherArray`. Но тогда возникает проблема: `anotherArray` содержит ссылки на `anotherObject` и `myObject`, так не нужно ли продублировать *и их*, вместо того чтобы ограничиться сохранением ссылок? Существование циклических ссылок приводит к проблеме бесконечного циклического дублирования объектов.

Нужно ли обнаружить циклическую ссылку и просто разбить циклический обход ссылок (оставляя глубокий элемент не полностью продублированным)? Выдать сообщение об ошибке? Выбрать некий промежуточный вариант?

Более того, не очень понятно, что должно считаться «дублированием» функции. Существуют разные трюки вроде извлечения сериализованного представления `toString()` исходного кода функции (который меняется в зависимости от реализации, и даже для разных ядер в зависимости от типа функции).

Как же ответить на все эти непростые вопросы? Разные фреймворки JS выбирают собственные интерпретации и принимают свои собственные решения. Но какое из них должно быть при-

нято в JS в качестве стандарта (и должно ли вообще). В течение долгого времени ясного ответа на эти вопросы не существовало.

Одно подмножество решений основано на том, что JSON-безопасные объекты (то есть объекты, которые могут быть сериализованы в строку JSON, которая затем разбирается в объект с той же структурой и значениями) легко дублируются следующим образом:

```
var newObj = JSON.parse( JSON.stringify( someObj ) );
```

Конечно, для этого необходимо убедиться в JSON-безопасности объекта. В одних ситуациях это делается тривиально, в других нет.

В то же время поверхностное копирование вполне понятно и создает меньше проблем, поэтому в ES6 для этой задачи была определена функция `Object.assign(..)`. `Object.assign(..)` получает в первом параметре объект-*приемник*, а в остальных параметрах — один или несколько объектов-*источников*. Функция перебирает все *перечисляемые* (см. следующий код), непосредственно существующие ключи объекта-источника и копирует их (посредством обычного присваивания `=`) в приемник. Также она возвращает приемник, как видно из следующего примера:

```
var newObj = Object.assign( {}, myObject );

newObj.a; // 2
newObj.b === anotherObject; // true
newObj.c === anotherArray; // true
newObj.d === anotherFunction; // true
```

В следующем разделе описываются «дескрипторы свойств» (характеристики свойств) и продемонстрировано использование `Object.defineProperty(..)`. Однако дублирование, которое выполняется для `Object.assign(..)`, ограничивается

простым присваиванием в стиле =, так что любые специальные характеристики свойств (например, возможность записи `writable`) объекта-источника не будут сохранены в объекте-приемнике.

Дескрипторы свойств

До выхода ES5 язык JavaScript не давал возможности напрямую анализировать характеристики свойств на программном уровне, например, узнать, доступно свойство только для чтения или нет.

Однако начиная с ES5 все свойства описываются *дескрипторами свойств*.

Рассмотрим следующий код:

```
var myObject = {
    a: 2
};

Object.getOwnPropertyDescriptor( myObject, "a" );
// {
//    value: 2,
//    writable: true,
//    enumerable: true,
//    configurable: true
// }
```

Как видите, дескриптор свойства (который в данном случае называется «дескриптором данных», поскольку он относится только к хранимому значению данных) для обычного свойства `a` содержит гораздо больше информации, чем значение `2`. Он также включает три других характеристики: `writable` (возможность записи), `enumerable` (перечисляемость) и `configurable` (возможность настройки).

Хотя вы можете просмотреть значения по умолчанию для характеристик дескриптора при создании обычного свойства, также можно воспользоваться вызовом `Object.defineProperty(..)` для добавления нового свойства или изменения существующего свойства (если оно допускает возможность настройки) с заданными характеристиками.

Пример:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: true,
    enumerable: true
} );

myObject.a; // 2
```

Вызовом `defineProperty(..)` мы вручную добавляем простое, обычное свойство `a` к объекту `myObject`. Тем не менее обычно ручное добавление свойств используется только в том случае, если вы хотите изменить нормальное поведение одной из характеристик дескриптора.

Writable

Характеристика `writable` определяет возможность изменения значения свойства.

Пример:

```
var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
```

```
    writable: false, // запись невозможна!
    configurable: true,
    enumerable: true
} );

myObject.a = 3;

myObject.a; // 2
```

Как видите, попытка изменения значения незаметно завершилась неудачей. Если попытаться сделать то же в режиме `strict`, произойдет ошибка:

```
"use strict";

var myObject = {};

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: false, // запись невозможна!
    configurable: true,
    enumerable: true
} );

myObject.a = 3; // TypeError
```

Ошибка `TypeError` сообщает, что изменить свойство с запретом записи невозможно.

Вскоре мы обсудим `get`/`set`-функции; а пока следует заметить, что `writable:false` означает, что значение не может быть изменено, что отчасти эквивалентно определению пустой функции записи. Чтобы пустая функция записи полностью соответствовала поведению `writable:false`, она должна при вызове выдавать ошибку `TypeError`.

Configurable

Если свойство допускает настройку, вы можете изменить определение его дескриптора при помощи функции `defineProperty(..)`:

```
var myObject = {
    a: 2
};

myObject.a = 3;
myObject.a; // 3

Object.defineProperty( myObject, "a", {
    value: 4,
    writable: true,
    configurable: false, // настройка невозможна!
    enumerable: true
} );

myObject.a; // 4
myObject.a = 5;
myObject.a; // 5

Object.defineProperty( myObject, "a", {
    value: 6,
    writable: true,
    configurable: true,
    enumerable: true
} ); // TypeError
```

При попытке изменить определение дескриптора свойства, для которого запрещена настройка, итоговый вызов `defineProperty(..)` приводит к ошибке `TypeError` независимо от режима `strict`. Будьте осторожны: как видите, переключение `configurable` в состояние `false` необратимо. Отменить его уже не удастся!

Следует помнить об одном нетривиальном исключении: даже если свойству уже присвоено `configurable:false`, характеристика `writable` всегда может быть переключена из `true` в `false` без выдачи ошибки, но ее не удастся вернуть в `true`, если она уже находится в состоянии `false`!

Также `configurable:false` блокирует возможность использования оператора `delete` для удаления существующего свойства:

```
var myObject = {
    a: 2
};

myObject.a; // 2
delete myObject.a;
myObject.a; // undefined

Object.defineProperty( myObject, "a", {
    value: 2,
    writable: true,
    configurable: false,
    enumerable: true
} );

myObject.a; // 2
delete myObject.a;
myObject.a; // 2
```

Как видите, последний вызов `delete` завершился неудачей (незаметно), потому что для свойства `a` запрещена настройка.

`delete` используется только для удаления свойств (тех, которые могут быть удалены) непосредственно из объекта. Если свойство объекта является последней оставшейся *ссылкой* на некоторый объект/функцию и для него будет вызван оператор `delete`, то это приведет к удалению ссылки, после чего объект/функция без ссылок может быть уничтожен уборщиком мусора. Тем не менее было бы неправильно рассматривать `delete` как инструмент осво-

бождения выделенной памяти, как в других языках (например, C/C++). `delete` — всего лишь операция удаления свойства объекта, ничего более.

Enumerable

Последняя характеристика дескриптора, которую мы здесь упомянем (есть еще две, о них будет рассказано позднее, при обсуждении get/set-функций), — `enumerable`.

Возможно, это очевидно следует из названия, но эта характеристика управляет тем, должно ли свойство включаться в перечисления свойств объектов, например, в циклах `for..in`. Если присвоить `enumerable` значение `false`, свойство не будет включаться в такие перечисления, хотя оно и остается полностью доступным. Со значением `true` свойство будет включаться в перечисления.

Для всех обычных свойств, определяемых пользователями, по умолчанию перечисление разрешено, так как чаще всего нужно именно это. Но если вы определяете специальное свойство, которое должно быть исключено из перечисления, установите для него `enumerable:false`.

Перечисление будет намного более подробно рассмотрено ниже, так что пока просто оставьте мысленную закладку на этой теме.

Неизменяемость

Иногда требуется создать свойства или объекты, которые не могут изменяться (случайно или намеренно). В ES5 была добавлена поддержка, позволяющая сделать это несколькими разными способами.

Важно заметить, что все эти способы создают поверхностную неизменяемость. Иначе говоря, они влияют только на объект и на

его непосредственные характеристики свойств. Если объект содержит ссылку на другой объект (массив, объект, функцию и т. д.), содержимое этого объекта остается изменяемым:

```
myImmutableObject.foo; // [1,2,3]
myImmutableObject.foo.push( 4 );
myImmutableObject.foo; // [1,2,3,4]
```

В этом фрагменте предполагается, что объект `myImmutableObject` уже создан и защищен как неизменяемый. Но чтобы также защитить содержимое `myImmutableObject.foo` (который сам является объектом — массивом), вы должны сделать неизменяемым `foo` одним или несколькими способами, представленными ниже.

Для программ JS нетипично создание неизменяемых объектов с глубокой вложенностью. Конечно, специальные ситуации встречаются, но в общем случае если у вас появляется желание зафиксировать и защитить от изменений все свои объекты, возможно, стоит сделать шаг назад и пересмотреть структуру своей программы, чтобы она была более устойчивой перед потенциальными изменениями в значениях объектов.

Объектные константы

Объединяя `writable:false` с `configurable:false`, можно фактически создать константу (с невозможностью изменения, переопределения или удаления) как свойство объекта:

```
var myObject = {};

Object.defineProperty( myObject, "FAVORITE_NUMBER", {
    value: 42,
    writable: false,
    configurable: false
} );
```

Запрет расширения

Если вы хотите запретить возможность добавления новых свойств в объект, но оставить остальные свойства объекта без изменений, вызовите `Object.preventExtensions(..)`:

```
var myObject = {
    a: 2
};

Object.preventExtensions( myObject );

myObject.b = 3;
myObject.b; // undefined
```

Если не действует режим `strict`, попытка создания `b` завершается неудачей без выдачи ошибки. В режиме `strict` выдается ошибка `TypeError`.

Seal

`Object.seal(..)` создает «запечатанный» объект; функция получает существующий объект и фактически вызывает для него `Object.preventExtensions(..)`, но также все существующие свойства получают пометку `configurable:false`.

Таким образом, к объекту не только нельзя добавлять новые свойства, но и также нельзя изменять конфигурацию или удалять существующие свойства (хотя вы можете изменять их значения).

Freeze

`Object.freeze(..)` создает «замороженный» объект; функция получает существующий объект и фактически вызывает для него `Object.seal(..)`, но также все свойства доступа к данным полу-

чают пометку `writable:false`, так что их значения не могут быть изменены.

Это самая высокая степень неизменяемости, которая может быть достигнута для самого объекта; она предотвращает любые изменения объекта или его непосредственных свойств (хотя, как упоминалось выше, не влияет на содержимое других объектов, на которые указывают ссылки).

Объект также можно отправить в «глубокую заморозку» — вызовите `Object.freeze(..)` для объекта, а затем рекурсивно переберите все объекты, на которые он ссылается (и которые до настоящего момента не изменялись), и вызовите `Object.freeze(..)` для этих объектов. Будьте осторожны, поскольку это может повлиять на другие (общие) объекты, которые вы, возможно, не собирались изменять.

[[Get]]

В механизме обращения к свойствам существует неочевидный, но важный нюанс. Возьмем следующий пример:

```
var myObject = {
    a: 2
};

myObject.a; // 2
```

`myObject.a` — обращение к свойству, но оно не ограничивается простым поиском свойства с именем `a` в объекте `myObject`, как могло бы показаться.

Согласно спецификации, этот фрагмент выполняет операцию `[[Get]]` (нечто вроде вызова функции: `[[Get]]()`) для `myObject`. Встроенная операция по умолчанию `[[Get]]` у объекта сначала

ищет в объекте свойство с указанным именем, и если оно будет найдено — возвращает соответствующее значение.

При этом алгоритм `[[Get]]` определяет другие важные аспекты поведения, если ему не удается найти свойство с указанным именем. О том, что при этом происходит (обход цепочки `[[Prototype]]`, если она существует), будет рассказано в главе 5.

У операции `[[Get]]` есть одна важная особенность: если ей не удается каким-либо образом получить значение запрашиваемого свойства, она вместо этого возвращает значение `undefined`:

```
var myObject = {
    a: 2
};

myObject.b; // undefined
```

Это поведение отличается от обращения к переменным по именам идентификаторов. При обращении к переменной, которую не удается разрешить соответствующим поиском по лексической области видимости, вы не получите результат `undefined`, как при обращении к свойствам объектов; вместо это будет выдана ошибка `ReferenceError`:

```
var myObject = {
    a: undefined
};

myObject.a; // undefined

myObject.b; // undefined
```

С точки зрения *значения* никакой разницы между этими двумя ссылками нет — обе дают результат `undefined`. Однако с незаметной операцией `[[Get]]` теоретически для ссылки `myObject.b` выполняется чуть больший объем работы, чем для ссылки `myObject.a`.

Если ограничиться анализом полученных значений, вам не удастся различить две ситуации: (1) свойство существует и содержит явное значение `undefined`, и (2) свойство не существует, а значение `undefined` было получено как возвращаемое значение по умолчанию, после того как операция `[[Get]]` не смогла вернуть более содержательное значение. Тем не менее вскоре вы увидите, что различить эти две ситуации *можно*.

[[Put]]

Раз существует внутренняя операция `[[Get]]` для получения значения из свойства, очевидно, должна существовать и операция `[[Put]]` по умолчанию.

Было бы заманчиво предположить, что при присваивании свойству объекта просто вызывается операция `[[Put]]` для записи или создания этого свойства в указанном объекте. Тем не менее ситуация не столь проста.

Поведение `[[Put]]` при вызове зависит от ряда факторов, в том числе (и прежде всего) от того, существует ли свойство в объекте или нет.

Если свойство присутствует, алгоритм `[[Put]]` в общих чертах проверяет следующее:

1. Является ли свойство дескриптором функции доступа (см. «Геттеры и сеттеры»)? В таком случае вызывается set-функция, если она существует.

2. Свойство является дескриптором данных с характеристикой `writable:false`? В таком случае при отсутствии режима `strict` происходит незаметный сбой, а в режиме `strict` выдается ошибка `TypeError`.

3. В остальных случаях значение присваивается существующему свойству, как обычно.

Если свойство отсутствует в объекте, операция [[Put]] становится еще более сложной и неочевидной. Мы вернемся к этой ситуации при рассмотрении [[Prototype]] в главе 10, чтобы она стала более понятной.

Геттеры и сеттеры

Операции [[Put]] и [[Get]] для объектов полностью управляют присваиванием значений существующим или новым свойствам, а также чтением из существующих свойств соответственно.

Будущие/расширенные возможности языка позволят переопределять операции [[Get]] или [[Put]] по умолчанию на уровне всего объекта (не только на уровне отдельных свойств). Данная тема выходит за рамки материала этой книги, но возможно, она будет рассмотрена в одной из будущих книг серии.

В ES5 появилась возможность переопределения некоторых операций по умолчанию не на уровне объектов, а на уровне отдельных свойств; для этой цели используются get- (геттеры) и set-методы (сеттеры). Геттер — свойства, которые вызывают скрытую функцию для получения нужного значения. Сеттер — свойства, которые вызывают скрытую функцию для присваивания значения.

Когда вы определяете для свойства геттер и/или сеттер, его определение становится «дескриптором методов доступа» (в отличие от «дескриптора данных»). Для дескрипторов методов доступа характеристики value и writable дескриптора считаются несущественными и игнорируются, а JS учитывает только характеристики set и get свойства (а также configurable и enumerable).

Пример:

```
var myObject = {
    // define a getter for `a`
    get a() {
        return 2;
    }
};

Object.defineProperty(
    myObject,   // приемник
    "b",        // имя свойства
    {           // дескриптор
        // определение геттера для `b`
        get: function(){ return this.a * 2 },
        // чтобы свойство `b` включалось в список свойств объекта
        enumerable: true
    }
);

myObject.a; // 2
myObject.b; // 4
```

При использовании синтаксиса объектных литералов как с `get a() { .. }`, так и с явным определением `defineProperty(..)`, в обоих случаях будет создано свойство объекта, которое на самом деле не содержит значения. Тем не менее при обращении к этому свойству автоматически вызывается скрытая функция для геттера, и возвращенное этой функцией значение становится результатом обращения к свойству:

```
var myObject = {
    // определение геттера для `a`
    get a() {
        return 2;
    }
};

myObject.a = 3;

myObject.a; // 2
```

Так как для a определен только геттер, при попытке задать значение a операция не выдаст ошибку, а просто игнорирует присваивание. Даже если бы существовал действительный сеттер, наш специализированный геттер запрограммирован так, что он возвращает только 2, так что попытка присваивания ни к чему не приведет.

Чтобы ситуация выглядела более разумно, свойства также могут определяться с сеттерами, переопределяющими операцию [[Put]] по умолчанию (то есть присваивание) на уровне отдельных свойств, как и следовало ожидать. Вам почти всегда следует объявлять как геттер, так и сеттер (определение только одного метода доступа часто приводит к неожиданному и нелогичному поведению):

```
var myObject = {
    // определить геттер для `a`
    get a() {
        return this._a_;
    },

    // определить сеттер для `a`
    set a(val) {
        this._a_ = val * 2;
    }
};

myObject.a = 2;

myObject.a; // 4
```

В этом примере заданное для присваивания значение 2 (операция [[Put]]) сохраняется в другой переменной _a_. Имя _a_ выбрано в данном примере исключительно по общепринятой схеме; оно не несет никакой информации о поведении — это обычное свойство, как и любое другое.

Существование

Ранее мы показали, что обращение к свойству (например, `myObject.a`) может привести к значению `undefined`, если это значение явно хранится в свойстве или же свойство вообще не существует. Итак, если значение в обоих случаях одинаково, то как же различить две ситуации?

Можно проверить, содержит ли объект некоторое свойство, *не запрашивая* значение этого свойства:

```
var myObject = {
    a: 2
};

("a" in myObject); // true
("b" in myObject); // false

myObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "b" ); // false
```

Оператор `in` проверяет, присутствует ли заданное свойство в объекте или на одном из более высоких уровней обхода цепочки `[[Prototype]]` (см. главу 10). С другой стороны, `hasOwnProperty(..)` только проверяет, присутствует ли свойство в объекте `myObject` или нет, и *не обращается* к цепочке `[[Prototype]]`. Мы вернемся к важным различиям между этими двумя операциями в главе 10, когда будем подробно рассматривать `[[Prototype]]`.

Функция `hasOwnProperty(..)` доступна для всех обычных объектов из-за делегирования `Object.prototype` (см. главу 10). Тем не менее возможно создать объект, который не связывается с `Object.prototype` (через `Object.create(null)` — см. главу 10). В этом случае вызов такого метода, как `myObject.hasOwnProperty(..)`, завершится неудачей.

В такой ситуации лучше воспользоваться более надежным способом выполнения проверки с `Object.prototype.hasOwnProperty.`

`call(myObject,"a")`, который заимствует базовый метод `hasOwnProperty(..)` и использует *явное связывание* (см. главу 7) для применения его к `myObject`.

Может показаться, что оператор `in` проверяет существование значения в контейнере, но в действительности он проверяет существование свойства с заданным именем. Это различие важно в отношении массивов, когда у разработчика возникает заманчивая мысль использовать оператор для проверки присутствия (например, 4 в [2, 4, 6]), но такая проверка не даст ожидаемого результата.

Перечисление

Ранее мы кратко рассмотрели концепцию «перечисляемости» при описании характеристики дескриптора свойств `enumerable`. Вернемся к ней и рассмотрим ее более подробно:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // разрешить для `a` перечисление, как обычно
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // ЗАПРЕТИТЬ для `b` перечисление
    { enumerable: false, value: 3 }
);

myObject.b; // 3
("b" in myObject); // true
myObject.hasOwnProperty( "b" ); // true
```

```
// .......

for (var k in myObject) {
    console.log( k, myObject[k] );
}
// "a" 2
```

Хотя свойство `myObject.b` существует, а его значение доступно, оно не отображается в цикле `for..in` (хотя, как ни странно, обнаруживается при проверке существования оператором `in`). Дело в том, что «перечисляемость» по сути означает «будет присутствовать в свойствах объекта при переборе».

Циклы `for..in` применительно к массивам могут давать довольно неожиданные результаты, поскольку в перечисление массива включаются не только все числовые индексы, но и все перечисляемые свойства. Циклы `for..in` рекомендуется использовать только с объектами, а для перебора массивов — традиционные циклы `for` с числовыми индексами.

Еще один способ различения перечисляемых и неперечисляемых свойств:

```
var myObject = { };

Object.defineProperty(
    myObject,
    "a",
    // свойство `a` является перечисляемым, как обычно
    { enumerable: true, value: 2 }
);

Object.defineProperty(
    myObject,
    "b",
    // `b` делается неперечисляемым
    { enumerable: false, value: 3 }
```

```
);

myObject.propertyIsEnumerable( "a" ); // true
myObject.propertyIsEnumerable( "b" ); // false

Object.keys( myObject ); // ["a"]
Object.getOwnPropertyNames( myObject ); // ["a", "b"]
```

`propertyIsEnumerable(..)` проверяет, существует ли свойство с заданным именем непосредственно у объекта, а также характеристику `enumerable:true`.

`Object.keys(..)` возвращает массив всех перечисляемых свойств, тогда как `Object.getOwnPropertyNames(..)` возвращает массив всех свойств (как перечисляемых, так и нет).

Если `in` отличается от `hasOwnProperty(..)` тем, обращаются они к цепочке `[[Prototype]]` или нет, `Object.keys(..)` и `Object.getOwnPropertyNames(..)` проверяют только заданный объект. Не существует (в настоящее время) встроенного механизма получения списка всех свойств, эквивалентного тому, что проверяет оператор `in` (перебор всех свойств для всех свойств цепочки `[[Prototype]]`, как объясняется в главе 10). Такой механизм можно смоделировать рекурсивным обходом цепочки `[[Prototype]]` для объекта с сохранением списка `Object.keys(..)` на каждом уровне — только перечисляемых свойств.

Перебор

Цикл `for..in` перебирает список перечисляемых свойств объекта (включая его цепочку `[[Prototype]]`). А если вы хотите перебирать значения, а не свойства?

Для массивов с числовым индексированием перебор значений обычно выполняется в стандартном цикле `for`:

```
var myArray = [1, 2, 3];

for (var i = 0; i < myArray.length; i++) {
    console.log( myArray[i] );
}
// 1 2 3
```

Здесь перебираются не значения, а индексы, которые используются для обращения к значениям (`myArray[i]`).

В ES5 также добавились некоторые вспомогательные средства перебора массивов, включая `forEach(..)`, `every(..)` и `some(..)`. Каждая вспомогательная функция получает функцию обратного вызова, которая будет применена к каждому элементу массива; различается только реакция на возвращаемое значение функции обратного вызова. `forEach(..)` перебирает все значения в массиве и игнорирует возвращаемые значения. `every(..)` продолжает перебор до конца или до того момента, когда обратный вызов вернет значение `false` (или «ложное» значение), а `some(..)` продолжает перебор до конца или до того момента, когда обратный вызов вернет значение `true` (или «истинное» значение). Специальные возвращаемые значения `every(..)` и `some(..)` отчасти напоминают команду `break` в обычном цикле `for`: они также могут прервать перебор до того, как он достигнет последнего элемента.

При переборе объекта в цикле `for..in` вы также получаете все значения косвенно, потому что в действительности перебор ведется только по перечисляемым свойствам объекта. Вам остается вручную обращаться к свойствам для получения значений.

В отличие от перебора индексов массива с числовым упорядочением (циклы `for` и другие итераторы), порядок перебора свойств объекта не гарантирован и может изменяться в зависимости от движка JS. Никогда не полагайтесь на наблюдаемый порядок чего-либо, что должно сохранять последовательность в разных средах, так как наблюдаемые закономерности ненадежны.

Но что, если вы хотите перебирать сами значения, а не индексы массива (и не свойства объекта)? К счастью, в ES6 появился синтаксис `for..of` для перебора массивов (и объектов, если объект определяет собственный итератор):

```
var myArray = [ 1, 2, 3 ];

for (var v of myArray) {
    console.log( v );
}
// 1
// 2
// 3
```

Цикл `for..of` запрашивает объект-итератор (у внутренней функции по умолчанию, которая в спецификации обозначается `@@iterator`) для того, что должно перебираться, а затем перебирает последовательные возвращаемые значения, полученные при вызове метода `next()` этого объекта-итератора (по одному для каждой итерации цикла).

У массивов имеется встроенная реализация `@@iterator`, так что `for..of` легко работает с ними (см. выше). Но чтобы лучше понять, как она работает, переберем массив вручную с использованием встроенной функции `@@iterator`:

```
var myArray = [ 1, 2, 3 ];
var it = myArray[Symbol.iterator]();

it.next(); // { value:1, done:false }
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { done:true }
```

Для получения доступа к внутреннему свойству `@@iterator` объекта используется специальное символическое имя (`Symbol`) ES6: Symbol.iterator. Семантика Symbol кратко упоминалась ранее в этой главе (см. «Вычисление имен свойств»), так что

> здесь действует та же логика. Для обращения к таким специальным свойствам всегда следует использовать символическое имя вместо специального значения, которое оно может содержать. Кроме того, несмотря на имя, `@@iterator` — не сам объект-итератор, а функция, которая возвращает объект-итератор — тонкая, но важная подробность!

Как показывает предыдущий фрагмент, возвращаемое значение вызова `next()` итератора представляет собой объект вида `{ value: .. , done: .. }`, где `value` — текущее значение итерации, а `done` — логическое значение, которое показывает, остались ли элементы для перебора.

Значение `3` было возвращено с признаком `done:false`, что выглядит немного странно. Вы должны вызвать `next()` в четвертый раз (что цикл `for..of` из предыдущего фрагмента делает автоматически), чтобы получить `done:true` и узнать, что перебор завершился. Причины этой странности выходят за рамки обсуждаемой темы, но они обусловлены семантикой функций-генераторов ES6.

Хотя массивы автоматически перебираются в циклах `for..of`, у обычных объектов нет встроенной функции `@@iterator`. Причины для этого намеренного недочета слишком сложны, чтобы обсуждать их здесь; скажу лишь, что в общем случае лучше не включать реализации, которые могут создать проблемы для будущих типов объектов.

Вы можете определить собственную реализацию `@@iterator` по умолчанию для любого объекта, который планируется использовать для перебора. Пример:

```
var myObject = {
    a: 2,
    b: 3
};
```

```
Object.defineProperty( myObject, Symbol.iterator, {
    enumerable: false,
    writable: false,
    configurable: true,
    value: function() {
        var o = this;
        var idx = 0;
        var ks = Object.keys( o );
        return {
            next: function() {
                return {
                    value: o[ks[idx++]],
                    done: (idx > ks.length)
                };
            }
        };
    }
} );

// ручной перебор `myObject`
var it = myObject[Symbol.iterator]();
it.next(); // { value:2, done:false }
it.next(); // { value:3, done:false }
it.next(); // { value:undefined, done:true }

// перебор `myObject` в `for..of`
for (var v of myObject) {
    console.log( v );
}
// 2
// 3
```

Мы использовали `Object.defineProperty(..)` для определения собственной реализации `@@iterator` (в основном для того, чтобы сделать ее неперечисляемой), но использование `Symbol` как вычисляемого имени свойства (см. ранее в этой главе) позволило бы объявить свойство напрямую, например, `var myObject = { a:2, b:3, [Symbol.iterator]: function() { /* .. */ } }`.

Каждый раз, когда цикл `for..of` вызывает `next()` для объекта-итератора `myObject`, внутренний указатель смещается и возвращает следующее значение из списка свойств объекта (см. врезку в этом разделе о порядке перебора свойств/значений объекта).

Мы только что рассмотрели простой перебор «значение за значением», но вы, конечно, можете определить для своих структур данных сколь угодно сложные правила перебора. Пользовательские итераторы в сочетании с циклом ES6 `for..of` — мощный новый синтаксический инструмент для работы с объектами, определяемыми пользователем.

Например, список объектов `Pixel` (со значениями координат `x` и `y`) может выбрать для упорядочения перебора критерий линейного расстояния от начала координат (0,0) или отфильтровать точки, находящиеся «слишком далеко», и т. д. Если только ваш итератор возвращает ожидаемое значение `{ value: .. }` при вызове `next()` и `{ done: true }` после завершения перебора, цикл ES6 `for..of` сможет выполнить перебор.

Вы даже можете определить «бесконечные» итераторы, которые никогда не «завершаются» и всегда возвращают новое значение (например, случайное число, увеличенное значение счетчика, уникальный идентификатор и т. д.). Впрочем, вы вряд ли будете использовать такие итераторы с бесконечным циклом `for..of`, так как он никогда не завершится, а ваша программа зависнет:

```
var randoms = {
    [Symbol.iterator]: function() {
        return {
            next: function() {
                return { value: Math.random() };
            }
        };
    }
};
```

```
var randoms_pool = [];
for (var n of randoms) {
    randoms_pool.push( n );

    // не продолжать бесконечно!
    if (randoms_pool.length === 100) break;
}
```

Этот итератор генерирует бесконечную последовательность случайных чисел, поэтому мы принимаем меры к тому, чтобы получить только 100 значений и избежать зависания программы.

Итоги

Объекты в JS могут определяться в литеральной форме (например, `var a = { .. }`) и в сконструированной форме (например, `var a = new Array(..)`). Литеральная форма почти всегда предпочтительна, но сконструированная форма в некоторых случаях предоставляет больше возможностей для создания объектов.

Многие люди ошибочно утверждают, что «в JavaScript нет ничего, кроме объектов», но это не так. Объекты — один из шести (или семи в зависимости от точки зрения) примитивных типов. У объектов есть подтипы (например, `function`), и они могут обладать специализированным поведением, например, `[object Array]` как внутренняя метка, представляющая подтип объекта-массива.

Объекты представляют собой коллекции пар «ключ/значение». К значениям можно обращаться как к свойствам в синтаксисе `.propName` или `["propName"]`. При каждом обращении к свойству движок вызывает внутреннюю операцию `[[Get]]` по умолчанию (`[[Put]]` для присваивания значений), которая не только ищет свойство непосредственно в объекте, но и обходит цепочку `[[Prototype]]` (см. главу 5), если свойство не было найдено.

Свойства обладают некоторыми характеристиками, которыми можно управлять при помощи дескрипторов свойств (например, `writable` и `configurable`). Кроме того, возможностью изменения объектов (и их свойств) можно управлять на разных уровнях при помощи `Object.preventExtensions(..)`, `Object.seal(..)` и `Object.freeze(..)`.

Свойства не обязательно содержат значения — они также могут быть «свойствами доступа» с геттерами/сеттерами. Также свойства могут быть или не быть *перечисляемыми*; например, эта характеристика управляет тем, будут ли они включаться в перебор в цикле `for..in`.

Также возможен перебор значений в структурах данных (массивы, объекты и т. д.) с использованием синтаксиса ES6 `for..of`. Для него необходим либо встроенный, либо пользовательский объект `@@iterator`, содержащий метод `next()` для последовательного перехода между значениями данных.

9 Классы

После знакомства с объектами из предыдущей главы вполне естественно перейти к объектно-ориентированному (ОО) программированию с классами. Сначала мы рассмотрим классово-ориентированный подход как паттерн проектирования, прежде чем изучать основные механизмы классов: создание экземпляров, наследование и (относительный) полиморфизм.

Вы увидите, что не все эти концепции имеют естественные аналоги в механизме объектов в JS, поэтому многие разработчики JavaScript применяют дополнительные средства (примеси и т. д.) для решения возникающих проблем.

В этой главе достаточно много места (вся первая половина) выделяется под теорию объектно-ориентированного программирования. Во второй половине эти концепции будут связаны с конкретным кодом JavaScript, когда речь пойдет о примесях. Сначала будет немало теоретических концепций и псевдокода, так что не унывайте, просто потерпите!

Теория классов

Классы/наследование описывают определенную форму организации кода и архитектуры — способ моделирования предметных областей реального мира в программных продуктах.

Объектно-ориентированное программирование (или классово-ориентированное) основано на том, что с данными по их природе связывается некоторое поведение (конечно, разное в зависимости от типа и природы данных), которое с этими данными работает. Таким образом, в процессе проектирования данные упаковываются (или инкапсулируются) вместе с поведением. В компьютерной теории эта концепция иногда называется *структурами данных*.

Например, последовательность символов, представляющих слово или фразу, обычно называется *строкой*. Символы представляют данные. Но вас почти всегда интересуют не столько сами данные, сколько возможность выполнения с ними *различных операций*, поэтому аспекты поведения, которые могут применяться к этим данным (вычисление длины, присоединение новых данных, поиск и т. д.) оформляются в виде методов класса `String`.

Любая конкретная строка представляет собой экземпляр этого класса — аккуратный пакет, состоящий как из символьных данных, так и из функциональности, которая к ним применяется.

Классы также подразумевают определенный механизм *классификации* структуры данных. Для этого любая заданная структура данных рассматривается как частный случай более общего базового определения.

А теперь проанализируем процесс классификации на часто приводимом примере. *Автомобиль* (car) может рассматриваться как

конкретная реализация более общего «класса» сущностей, который можно назвать «транспортным средством» (vehicle).

Для моделирования этой связи классами на программном уровне мы определим классы `Vehicle` и `Car`.

Определение `Vehicle` может включать такие характеристики, как способность к передвижению, возможность перевозить людей и т. д. — все это относится к поведению. В `Vehicle` определяется все то, что является общим для всех (или очень многих) разных видов транспорта (самолеты, поезда, автомобили).

Было бы неразумно переопределять базовое поведение «возможность перевозить людей» снова и снова для всех разных типов транспортных средств. Вместо этого данная способность определяется один раз в `Vehicle`, а затем при определении `Car` мы просто указываем, что этот класс *наследует* от базового определения из `Vehicle` (или *расширяет* его). Определение `Car` специализирует общее определение `Vehicle`.

Хотя `Vehicle` и `Car` совместно определяют поведение посредством методов, данными экземпляра будут такие атрибуты, как идентификационный номер транспортного средства и т. д.

Другая ключевая концепция классов — *полиморфизм* — описывает концепцию того, что общее поведение родительского класса может переопределяться в дочернем классе для определения более конкретной реализации. Относительный полиморфизм позволяет обращаться к базовому поведению из переопределенного поведения.

Из теории классов следует, что родительский и дочерний классы используют одно имя метода для некоторого аспекта поведения, так что дочерний класс переопределяет родителя (выборочно). Как вы вскоре увидите, в коде JavaScript это с большой вероятностью приводит к неприятностям и снижает надежность кода.

Паттерн проектирования «класс»

Возможно, вы никогда не рассматривали классы как паттерн проектирования. Чаще обсуждаются такие популярные паттерны ОО-проектирования, как «итератор», «наблюдатель», «фабрика», «одиночный объект» и т. д. Такое представление почти предполагает, что ОО-классы относятся к низкоуровневым механизмам, которые реализуют все (высокоуровневые) паттерны проектирования, словно ОО-программирование закладывает фундамент для всего (обычного) кода.

В зависимости от уровня вашей теоретической подготовки в программировании вы, возможно, слышали о *процедурном программировании* как о способе описания кода, который состоит только из процедур (то есть функций), вызывающих другие функции без более высоких абстракций. И может быть, вас учили, что классы должны применяться для преобразования «спагетти-кода» в процедурном стиле в хорошо структурированный, четко организованный код.

Конечно, если у вас имеется опыт *функционального программирования* (монады и т. д.), вы отлично знаете, что классы — всего лишь один из нескольких распространенных паттернов программирования. Но другие, возможно, впервые зададут себе вопрос, действительно ли классы являются фундаментальной основой для кода или же это необязательная абстракция, накладываемая на код.

Некоторые языки (такие, как Java) не дают выбора, так что здесь никакой «необязательности» нет — вы работаете только с классами и ничем более. Другие языки — такие, как C/C++ или PHP, — предоставляют как процедурный синтаксис, так и синтаксис, ориентированный на классы, так что выбор стиля или комбинации стилей остается на усмотрение разработчика.

«Классы» JavaScript

Какое положение здесь занимает JavaScript? Синтаксические элементы, *сходные* с элементами классов (например, `new` и `instanceof`), существуют в JS уже не первый день, а в ES6 появились такие дополнения, как ключевое слово `class` (см. приложение Г).

Но означает ли это, что в JavaScript действительно *существуют* классы? Ответ прост и однозначен: НЕТ.

Поскольку классы являются паттерном проектирования, вы *можете* с некоторыми усилиями (как будет показано в оставшейся части этой главы) приближенно реализовать большую часть классической функциональности классов. JS пытается удовлетворить в высшей степени распространенную *потребность* в классах, для чего он предоставляет синтаксис в стиле классов.

Несмотря на внешнее сходство синтаксиса с классами, вся механика JavaScript словно сопротивляется использованию *паттерна «класс»*, потому что те механизмы, на основе которых строятся эти возможности, работают совершенно иначе. Синтаксические удобства и (очень часто используемые) библиотеки «классов» JavaScript очень стараются скрыть от вас реальное положение дел, но рано или поздно вы столкнетесь с тем фактом, что *классы* других языков имеют мало общего с «классами», имитируемыми в JS.

Все это сводится к тому, что классы являются необязательным паттерном при проектировании программного обеспечения, и вы можете выбрать, использовать их в JavaScript или нет. Поскольку многие разработчики сильно расположены к проектированию, ориентированному на классы, в оставшейся части этой главы мы исследуем, что нужно делать для поддержания иллюзии классов средствами JS и с какими проблемами вы столкнетесь.

Механика классов

Во многих языках, ориентированных на классы, «стандартная библиотека» предоставляет структуру данных стека (операции занесения, извлечения и т. д.) в виде класса `Stack`. Этот класс содержит внутренний набор переменных для хранения данных, а также предоставляет набор общедоступных аспектов поведения («методов»), предоставляющих вашему коду средства для взаимодействия с (скрытыми) данными (добавление и удаление данных и т. д.).

Но в таких языках вы не работаете с `Stack` напрямую (если только не создать ссылку на статический компонент класса, но эта тема выходит за рамки нашего обсуждения). Класс `Stack` — всего лишь абстрактное объяснение того, что должен делать любой стек, но сам по себе «стеком» он не является. Чтобы иметь конкретную структуру данных, с которой можно работать, необходимо создать экземпляр класса `Stack`.

Строительство

Традиционная метафора для мышления, основанного на «классах» и «экземплярах», происходит из архитектуры и строительства.

Архитектор планирует все характеристики здания: ширину, высоту, количество окон и их расположение, и даже материал, который должен использоваться для стен и крыши. Вполне возможно, что на этот момент его не интересует, где будет строиться это здание или сколько его копий будет построено.

Архитектора также не интересует содержимое здания (мебель, обои, вентиляторы и т. д.) — только тип структуры, в которой все это будет содержаться.

Архитектурные чертежи — всего лишь *план* здания. Они не образуют здание, в котором можно ходить, сидеть и т. д. Для этой

задачи нужен строитель. Строитель берет план и точно следует ему при строительстве здания. Можно с полным основанием сказать, что он копирует предполагаемые характеристики из плана в физическое здание.

После того как здание будет построено, оно становится физическим воплощением плана из чертежа — хочется верить, идеально воплощенной *копией*. Тогда строитель переходит на пустое место где-то в окрестностях и повторяет все заново, создавая еще одну копию.

Связь между зданием и планом не прямая. Вы можете проанализировать план, чтобы понять, какую структуру имеет здание (для любых частей, для которых недостаточно изучить само здание). Но если вы захотите открыть дверь, придется идти к самому зданию — на плане всего лишь нарисованы линии, представляющие местоположение двери.

Класс — тот же план. Чтобы получить объект, с которым можно взаимодействовать, необходимо *создать экземпляр* (построить) на основе класса. Конечным результатом такого построения становится объект, обычно называемый *экземпляром*. Вы можете напрямую вызывать методы экземпляра и обращаться к его любым открытым свойствам по мере надобности.

Этот объект является копией характеристик, описываемых классом.

Когда вы заходите в здание, вряд ли вы найдете в нем копию плана, по которому это здание строилось, — скорее всего, план будет храниться где-нибудь в архиве. Точно так же экземпляр обычно не может использоваться для прямого обращения к его классу или манипуляций с ним, но по крайней мере вы сможете определить, на базе какого класса был создан этот экземпляр.

Полезнее рассматривать прямую связь класса с экземпляром объекта вместо косвенной связи между экземпляром объекта и клас-

сом, на базе которого он был создан. Класс воплощается в форме объекта операцией копирования (рис. 9.1).

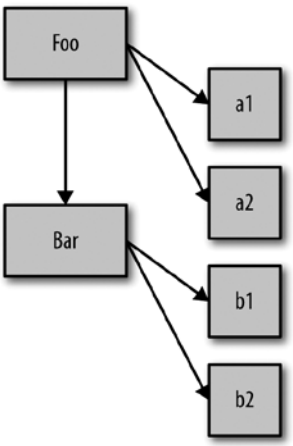

Рис. 9.1

Как видите, стрелки направлены слева направо и сверху вниз; они обозначают операции копирования (как концептуальные, так и физические).

Конструктор

Экземпляры классов конструируются специальным методом класса, имя которого обычно совпадает с именем класса. Этот метод называется *конструктором*. Его непосредственная задача — инициализация любой информации (состояния), необходимой экземпляру.

Для примера рассмотрим следующий неформальный псевдокод (вымышленный синтаксис) для классов:

```
class CoolGuy {
    specialTrick = nothing

    CoolGuy( trick ) {
        specialTrick = trick
    }

    showOff() {
        output( "Here's my trick: ", specialTrick )
    }
}
```

Чтобы создать экземпляр `CoolGuy`, следует вызвать конструктор класса:

```
Joe = new CoolGuy( "jumping rope" )

Joe.showOff() // Here's my trick: jumping rope
```

Обратите внимание: класс `CoolGuy` имеет конструктор `CoolGuy()`, который вызывается в выражении `new CoolGuy(..)`. Конструктор возвращает объект (экземпляр класса), и для этого объекта можно вызвать метод `showOff()`, который выводит информацию для конкретного экземпляра `CoolGuy`.

Конструктор класса *принадлежит* классу, а его имя почти всегда совпадает с именем класса. Кроме того, конструкторы практически всегда должны вызываться с оператором `new`, чтобы языковое ядро понимало, что вы хотите сконструировать новый экземпляр класса.

Наследование

В языках, ориентированных на использование классов, вы можете не только определить класс для создания экземпляров, но и определить другой класс, наследующий от первого.

Второй класс также называется «дочерним классом», а первый — «родительским классом». Разумеется, эти термины происходят от метафоры родителей и потомков, хотя, как вы вскоре увидите, метафора несколько притянута.

Когда у родителя имеется биологический потомок, генетические характеристики родителя копируются в потомка. Разумеется, в большинстве репродуктивных систем существуют два родителя, которые вносят в комбинацию генов равный вклад. Но для целей нашей метафоры будем предполагать, что родитель только один.

Появившийся на свет потомок отделен от родителя. Наследие родителя оказало на него серьезное влияние, однако потомок — вполне неповторимая и самостоятельная личность. Если у ребенка рыжие волосы, это не означает, что волосы родителя *были* или обязательно *станут* рыжими.

Аналогичным образом дочерний класс после определения существует отдельно от родительского. Дочерний класс содержит исходную копию поведения родителя, но он может переопределять любое унаследованное поведение и даже определять новые аспекты поведения.

Важно помнить, что речь идет о родительском и дочернем классах, которые не являются физическими объектами. Именно здесь метафора родителей/потомков начинает создавать путаницу, потому что на самом деле родительский класс скорее напоминает ДНК родителя, а дочерний класс — ДНК потомка. Чтобы появился человек, с которым можно было бы пообщаться, необходимо создать экземпляр на основе каждого набора ДНК. Но давайте отложим биологических родителей и потомков и рассмотрим наследование под несколько иным углом: возьмем разные типы транспортных средств. Кстати, это одна из самых канонических (и уже набивших оскомину) метафор для понимания наследования.

Вернемся к обсуждению `Vehicle` и `Car` из более раннего примера этой главы. Рассмотрим следующий неформальный псевдокод (вымышленный синтаксис) для наследования классов:

```
class Vehicle {
    engines = 1

    ignition() {
        output( "Turning on my engine." );
    }

    drive() {
        ignition();
        output( "Steering and moving forward!" )
    }
}

class Car inherits Vehicle {
    wheels = 4

    drive() {
        inherited:drive()
        output( "Rolling on all ", wheels, " wheels!" )
    }
}

class SpeedBoat inherits Vehicle {
    engines = 2

    ignition() {
        output( "Turning on my ", engines, " engines." )
    }

    pilot() {
        inherited:drive()
        output( "Speeding through the water with ease!" )
    }
}
```

Класс `Vehicle` определяется с одним двигателем, операцией включения зажигания и операцией перемещения. Но вам никак не удастся произвести обобщенное «транспортное средство», так что на данный момент это скорее абстрактная концепция.

Для ясности и компактности конструкторы этих классов были опущены.

Соответственно мы определяем две конкретные разновидности транспортных средств: `Car` (автомобиль) и `SpeedBoat` (катер). Каждая разновидность наследует общие характеристики `Vehicle`, но эти характеристики специализируются для каждой разновидности. У автомобиля четыре колеса, а у катера — два двигателя; это означает, что для включения зажигания на обоих двигателях придется принять особые меры.

Полиморфизм

`Car` определяет собственный метод `drive()`, который переопределяет одноименный метод, унаследованный от `Vehicle`. Но метод `drive()` специализации `Car` вызывает `inherited:drive()`; это означает, что `Car` может обратиться к исходной унаследованной версии `drive()` до ее переопределения. Метод `pilot()` специализации `SpeedBoat` также обращается к своей унаследованной копии `drive()`.

Этот прием называется *полиморфизмом* (или *виртуальным полиморфизмом*). В контексте нашего текущего обсуждения назовем его *относительным полиморфизмом*.

Полиморфизм — намного более обширная тема, чтобы рассматривать ее здесь, но текущая «относительная» семантика указывает

на один конкретный аспект: идею о том, что любой метод может обратиться к другому методу (с тем же или другим именем) на более высоком уровне иерархии наследования. Мы используем термин «относительный», потому что вместо абсолютного определения нужного уровня наследования (то есть класса) мы ссылаемся на него относительно, фактически говоря: «обратиться на один уровень вверх».

Во многих языках вместо `inherited:` из примера используется ключевое слово `super`; ведь родителя/предка текущего класса часто называют «суперклассом».

У полиморфизма есть еще один аспект: имя метода может иметь несколько определений на разных уровнях цепочки наследования, и эти определения автоматически выбираются при поиске вызываемых методов.

Два примера такого поведения встречаются в предыдущем примере: `drive()` определяется как в `Vehicle`, так и в `Car`, а `ignition()` определяется в `Vehicle` и `SpeedBoat`.

Другая возможность, которую предоставляют традиционные языки, ориентированные на использование классов, — применение super в конструкторе дочернего класса для обращения к конструктору родительского класса. Дело в том, что с настоящими классами конструктор принадлежит классу. В JS все наоборот — правильнее считать, что «класс» принадлежит конструктору (ссылки на типы `Foo.prototype...`). Так как в JS связь между родителем и потомком существует только между двумя объектами .prototype в соответствующих конструкторах, сами конструкторы не связаны напрямую, и не существует простого способа относительного обращения из одного к другому (см. приложение Г с описанием синтаксиса `class` в ES6, «решающего» эту проблему при помощи ключевого слова `super`).

Интересное последствие полиморфизма встречается в `ignition()`. Внутри `pilot()` происходит относительно-полиморфное обращение к «унаследованной» версии `drive()` специализации `Vehicle`. Но `drive()` обращается к методу `ignition()` просто по имени (без относительной ссылки).

Какую версию `ignition()` будет использовать языковое ядро — из `Vehicle` или из `SpeedBoat`? Оно использует версию `ignition()` из `SpeedBoat`. Если вы создадите экземпляр самого класса `Vehicle`, а затем вызовете его версию `drive()`, языковое ядро вместо этого использует определение метода `ignition()` из `Vehicle`.

Иначе говоря, определение метода `ignition()` *полиморфно изменяется* в зависимости от того, к какому классу (уровню наследования) относится экземпляр.

На первый взгляд это может показаться слишком отвлеченной теоретической подробностью. Но понимание таких подробностей необходимо для сравнения похожих (но на самом деле различных) аспектов поведения механизма `[[Prototype]]` в JavaScript.

При наследовании сами классы (не экземпляры объектов, созданные на их основе) могут использовать *относительные* обращения к классу, от которого они наследуют. Такие относительные ссылки обычно называются `super`.

Помните приведенную выше диаграмму? (См. рис. 9.1.)

Обратите внимание: как при создании экземпляра (`a1`, `a2`, `b1` и `b2`), так и при наследовании (`Bar`) стрелки обозначают операцию копирования.

На концептуальном уровне может показаться, что дочерний класс `Bar` может обращаться к поведению своего родительского класса `Foo` по относительной полиморфной ссылке (то есть `super`). Однако в действительности дочерний класс всего лишь получает копию унаследованного поведения от своего родительского класса.

Если дочерний класс «переопределяет» унаследованный метод, на самом деле поддерживаются обе версии метода, исходная и переопределенная, так что обе они доступны. Пусть полиморфизм не создает у вас обманчивого впечатления, будто дочерний класс связан с родительским. Дочерний класс всего лишь получает копию нужных компонентов от родительского класса. Наследование означает копирование.

Множественное наследование

Помните предшествующее обсуждение родителей, потомков и ДНК? Тогда я сказал, что метафора несовершенна, потому что с биологической точки зрения большинство потомства происходит от двух родителей. Если бы класс мог наследовать от двух других классов, он бы лучше подходил под метафору «родитель/потомок».

Некоторые объектно-ориентированные языки позволяют указать более одного «родительского» класса для наследования. Множественное наследование означает, что каждое определение родительского класса копируется в дочерний класс.

На первый взгляд может показаться, что это мощное расширение программирования, ориентированного на использование классов, поскольку оно дает возможность объединить больший объем функциональности. Однако при этом возникает целый ряд непростых вопросов. Если оба родительских класса предоставляют метод с именем `drive()`, то какую версию следует использовать по ссылке `drive()` в дочернем классе? Всегда вручную указывать, какой именно родитель имеется в виду? Но тогда теряется часть элегантности полиморфного наследования.

Также существует проблема *ромбовидного наследования* — ситуация, в которой дочерний класс D наследует от двух родительских клас-

сов (B и C), каждый из которых в свою очередь наследует от общего предка A. Если A предоставляет метод `drive()`, а оба класса B и C переопределяют этот метод, то какую версию следует использовать при обращении к `drive()` из D (`B:drive()` или `C:drive()`) (рис. 9.2)?

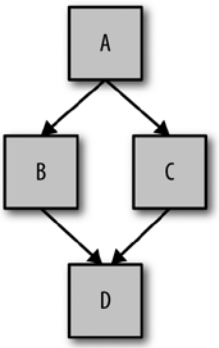

Рис. 9.2

Эти сложности далеко не ограничиваются этим беглым упоминанием. Я говорю о них только для того, чтобы вы могли сравнить ситуацию с механизмами JavaScript.

В JavaScript все просто: встроенный механизм «множественного наследования» отсутствует. Многие считают, что это хорошо, потому что снижение сложности более чем компенсирует «сокращение» функциональности. Тем не менее, как вы вскоре увидите, это не мешает разработчикам пытаться моделировать множественное наследование различными способами.

Примеси

Объектный механизм JavaScript не выполняет *автоматического* копирования при наследовании или создании экземпляров.

В JavaScript вообще не существует «классов», экземпляры которых можно было бы создавать; есть только объекты. А объекты не копируются в другие объекты, они *связываются* между собой (подробнее об этом в главе 10).

Так как наблюдаемое поведение классов в других языках подразумевает копирование, для начала посмотрим, как разработчики JS имитируют *отсутствующее* поведение копирования классов в JavaScript: они используют *примеси* (mixins). Мы рассмотрим два типа примесей, *явные* и *неявные*.

Явные примеси

Снова вернемся к примеру с `Vehicle` и `Car`. Так как JavaScript не обеспечивает автоматического копирования поведения из `Vehicle` в `Car`, вместо этого можно создать функцию, которая будет выполнять копирование вручную. Такая функция во многих библиотеках/фреймворках часто называется `extend(..)`, но мы назовем ее `mixin(..)` для демонстрации:

```
// сильно упрощенный пример `mixin(..)`:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // копировать, если не существует
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}

var Vehicle = {
    engines: 1,
    ignition: function() {
        console.log( "Turning on my engine." );
    },
```

```
    drive: function() {
        this.ignition();
        console.log( "Steering and moving forward!" );
    }
};

var Car = mixin( Vehicle, {
    wheels: 4,

    drive: function() {
        Vehicle.drive.call( this );
        console.log(
           "Rolling on all " + this.wheels + " wheels!"
        );
    }
} );
```

Неочевидная, но важная подробность: речь идет уже не о классах, потому что в JavaScript нет классов. Vehicle и Car — всего лишь объекты, из которых (и в которые) соответственно копируется функциональность.

`Car` теперь содержит копию свойств и функций из `Vehicle`. С технической точки зрения сами функции не дублируются; копируются только ссылки на функции. Таким образом, `Car` теперь содержит свойство с именем `ignition`, которое представляет собой скопированную ссылку на функцию `ignition()`, а также свойство `engines` со скопированным значением `1` из `Vehicle`.

`Car` уже содержит свойство `drive` (функция), так что ссылка на свойство не переопределяется (см. команду `if` в `mixin(..)` выше).

Снова о полиморфизме

Рассмотрим команду `Vehicle.drive.call( this )`. Я называю такую конструкцию *явным псевдополиморфизмом*. Вспомните, что в при-

веденном выше псевдокоде эта строка имела вид `inherited:drive()`, что называлось *относительным полиморфизмом*.

В JavaScript отсутствует поддержка относительного полиморфизма (до ES6; см. приложение Г). Поскольку `Car` и `Vehicle` содержат функции с тем же именем `drive()`, чтобы отличать один вызов от другого, потребуется абсолютная (не относительная) ссылка. Мы явно задаем объект `Vehicle` по имени и вызываем для него функцию `drive()`.

Но если мы используем запись `Vehicle.drive()`, `this` для этой функции будет связываться с объектом `Vehicle` вместо объекта `Car` (см. главу 7), а это не то, чего мы добивались. По этой причине мы используем `.call( this )` (глава 7), для того чтобы функция `drive()` выполнялась в контексте объекта `Car`.

Если бы идентификатор функции `Car.drive()` не перекрывался («замещение»; см. главу 11) с `Vehicle.drive()`, полиморфизм методов не наблюдался бы. Таким образом, ссылка на `Vehicle.drive()` была бы заменена вызовом `mixin(..)`, и мы могли напрямую обратиться к `this.drive()`. Замещение идентификаторов — причина, по которой нам приходится использовать более сложное решение с явным полиморфизмом.

В языках, ориентированных на использование классов, в которых поддерживается относительный полиморфизм, связь между `Car` и `Vehicle` устанавливается один раз — в начале определения класса, так что такие связи отслеживаются только в одном месте.

Но из-за особенностей JavaScript явный псевдополиморфизм (из-за замещения) создает ненадежное ручное/явное связывание в каждой функции, в которой потребуются такие (псевдо)полиморфные ссылки. Это может существенно повысить затраты на

сопровождение. Более того, хотя явный псевдополиморфизм может эмулировать поведение множественного наследования, он только повышает сложность и непрочность. Обычно такие решения приводят к появлению более сложного, неудобочитаемого и сложного в сопровождении кода. Явного псевдополиморфизма следует избегать, насколько это возможно, потому что во многих отношениях потери перевешивают пользу.

Примесное копирование

Вспомните приведенную выше функцию `mixin(..)`:

```
// сильно упрощенный пример `mixin()`:
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        // копировать, если не существует
        if (!(key in targetObj)) {
            targetObj[key] = sourceObj[key];
        }
    }

    return targetObj;
}
```

Посмотрим, как работает функция `mixin(..)`. Она перебирает свойства `sourceObj` (`Vehicle` в данном случае), и если в `targetObj` (`Car` в нашем примере) нет подходящего свойства с заданным именем — создает копию. Поскольку копия создается при уже существующем исходном объекте, необходимо действовать внимательно, чтобы случайно не заменить приемное свойство.

Если бы копирование было выполнено сначала, перед заданием содержимого, специфического для `Car`, проверку на `targetObj` можно было бы опустить, но такое решение более громоздко и менее эффективно, так что в общем случае оно считается менее предпочтительным:

```
// альтернативная версия mixin, менее "безопасная" для перезаписи
function mixin( sourceObj, targetObj ) {
    for (var key in sourceObj) {
        targetObj[key] = sourceObj[key];
    }

    return targetObj;
}

var Vehicle = {
    // ...
};

// сначала создать пустой объект, в который копируется
// содержимое из Vehicle
var Car = mixin( Vehicle, { } );

// теперь скопировать предполагаемое содержимое в Car
mixin( {
    wheels: 4,

    drive: function() {
        // ...
    }
}, Car );
```

При любом подходе происходит явное копирование неперекрывающегося содержимого `Vehicle` в `Car`. Название «примесь» (mixin) происходит от альтернативного способа изложения задачи: содержимое `Vehicle` «примешивается» к `Car` подобно тому, как шоколадная крошка примешивается в песочное тесто.

В результате операции копирования `Car` начинает существовать отдельно от `Vehicle`. Если вы добавите свойство в `Car`, это никак не повлияет на `Vehicle`, и наоборот.

Так как два объекта также содержат общие ссылки на общие функции, это означает, что даже ручное копирование функций (то есть примесей) из одного объекта в другой не моделирует

реального копирования из класса в экземпляр, которое происходит в языках, ориентированных на использование классов.

Здесь были пропущены некоторые второстепенные подробности. Два объекта могут «влиять» друг на друга некоторыми неочевидными способами, например, если оба они содержат ссылку на общий объект (скажем, массив).

Функции JavaScript на самом деле не могут дублироваться (стандартным, надежным образом), так что в итоге происходит *дублирование ссылки* на один общий объект функции (функции являются объектами; см. главу 8). Если вы измените один из общих объектов функций (например, `ignition()`), добавив к нему свойства, это отразится через общую ссылку как на `Vehicle`, так и на `Car`.

Явные примеси — полезный механизм JavaScript. Тем не менее они кажутся более мощными, чем в действительности. Копирование свойства из одного объекта в другой приносит меньше *реальной* пользы, чем повторное определение свойств (по одному разу для каждого объекта). И это особенно справедливо с учетом упомянутого нюанса со ссылками на объекты функций.

Явные примеси двух и более объектов к объекту-приемнику позволяют частично моделировать поведение множественного наследования, однако не существует прямых средств разрешения конфликтов, если один и тот же метод или свойство копируется более чем из одного источника. Некоторые разработчики/библиотеки предлагают методы «позднего связывания» и другие экзотические обходные решения, но по сути такие «трюки» *обычно* лишь увеличивают объем работы (с меньшим быстродействием), а не сокращают его.

Будьте осторожны и используйте явные примеси там, где они действительно делают код более понятным. Избегайте этого пат-

терна, если он усложняет трассировку кода или создает излишние, громоздкие зависимости между объектами.

Если правильно работать с примесями становится *труднее*, чем без них, вероятно, вам стоит отказаться от использования примесей. Если вам приходится использовать сложные библиотеки/вспомогательные функции, чтобы проработать все детали, это может свидетельствовать о том, что вы только усложняете свою задачу — скорее всего, без всякой необходимости. В главе 11 вы узнаете, как проще достичь желаемой цели без лишних хлопот.

Паразитическое наследование

Разновидность паттерна явной примеси — явная в одних отношениях и неявная в других — называется «паразитным наследованием». Ее популяризацией занимается в основном Дуглас Крокфорд.

Вот как она может работать:

```
// "Традиционный класс JS" `Vehicle`
function Vehicle() {
    this.engines = 1;
}

Vehicle.prototype.ignition = function() {
    console.log( "Turning on my engine." );
};

Vehicle.prototype.drive = function() {
    this.ignition();
    console.log( "Steering and moving forward!" );
};

// "Паразитический класс" `Car`
function Car() {
    // сначала `car` является `Vehicle`
```

```
    var car = new Vehicle();

    // изменить `car` для создания специализации
    car.wheels = 4;

    // сохранить привилегированную ссылку на `Vehicle::drive()`
    var vehDrive = car.drive;

    // переопределить `Vehicle::drive()`
    car.drive = function() {
        vehDrive.call( this );
        console.log(
           "Rolling on all " + this.wheels + " wheels!"
        );

    return car;
}

var myCar = new Car();
myCar.drive();
// Turning on my engine.
// Steering and moving forward!
// Rolling on all 4 wheels!
```

Как видно из листинга, мы сначала копируем определение из родительского класса (объекта) `Vehicle`, а затем примешиваем определение дочернего класса (объекта) (с сохранением привилегированных ссылок на родительский класс при необходимости) и передаем сформированный объект `car` как дочерний экземпляр.

При вызове `new Car()` создается новый объект, на который указывает ссылка `this` из `Car`. Но поскольку мы не используем этот объект, а вместо этого возвращаем собственный объект car, исходный созданный объект просто теряется. Таким образом, `Car()` можно вызвать без ключевого слова new — функциональность останется той же, но обойдется без лишних операций создания объекта/уборки мусора.

Неявные примеси

Неявные примеси тесно связаны с явным псевдополиморфизмом (см. выше). А это означает, что им присущи свои потенциальные риски и проблемы.

Рассмотрим следующий пример:

```
var Something = {
    cool: function() {
        this.greeting = "Hello World";
        this.count = this.count ? this.count + 1 : 1;
    }
};

Something.cool();
Something.greeting; // "Hello World"
Something.count; // 1

var Another = {
    cool: function() {
        // неявная примесь `Something` к `Another`
        Something.cool.call( this );
    }
};

Another.cool();
Another.greeting; // "Hello World"
Another.count; // 1 (не использует общее состояние с `Something`)
```

В вызове `Something.cool.call( this )`, который может произойти либо при вызове конструктора (самый распространенный случай), либо при вызове метода (как здесь), мы фактически «заимствуем» функцию `Something.cool()` и вызываем ее в контексте `Another` (через связывание `this`) вместо `Something`. В результате присваивания, выполняемые `Something.cool()`, применяются к объекту `Another` вместо объекта `Something`.

Итак, говорят, что поведение `Something` «примешивается» к `Another`.

Хотя этот прием вроде бы использует функциональность повторного связывания `this`, он содержит ненадежный вызов `Something.cool.call( this )`, который невозможно преобразовать в относительную (и потому более гибкую) ссылку и к которому следует относиться с осторожностью. В общем случае таких конструкций лучше избегать, чтобы код был более понятным и создавал меньше проблем с сопровождением.

Итоги

Классы являются паттерном проектирования. Многие языки предоставляют синтаксис, который делает возможным естественное проектирование программного кода, ориентированное на использование классов. В JS также имеется похожий синтаксис, но по своему поведению он сильно отличается от того, к чему вы привыкли с классами из других языков.

Классы подразумевают копирование.

При создании экземпляра традиционных классов происходит копирование поведения из класса в экземпляр. При наследовании классов также происходит копирование поведения из родителя в потомка.

Может показаться, что полиморфизм (существование разных функций с одинаковыми именами на разных уровнях цепочки наследования) подразумевает наличие относительной связи по ссылке от потомка к родителю, но в действительности он всего лишь является результатом копирования.

JavaScript не использует автоматическое копирование (как обычно подразумевают классы) между объектами.

Паттерн «примесь» (явный и неявный) часто используется для моделирования поведения копирования классов, но обычно он ведет к уродливому и ненадежному синтаксису, например явному псевдополиморфизму (`OtherObj.methodName.call(this, ...)`), что часто приводит к созданию кода, более сложного для понимания и сопровождения.

Явные примеси тоже не совсем точно воспроизводят поведение копирования классов, так как для объектов (и функций) копируются только общие ссылки, а не сами объекты/функции. На такие нюансы необходимо обращать внимание, иначе они станут источником всевозможных проблем.

В целом имитация классов в JS обычно готовит больше скрытых ловушек в будущем, чем решает *реальных* проблем в настоящем.

10 Прототипы

В главах 8 и 9 я несколько раз упоминал цепочку `[[Prototype]]`, но не объяснял, что же это такое. В этой главе прототипы будут рассмотрены более подробно.

Все попытки имитации поведения копирования классов, описанные в главе 9 и названные разновидностями примесей, полностью обходят механизм цепочки `[[Prototype]]`, которому посвящена эта глава.

[[Prototype]]

У объектов JavaScript имеется внутреннее свойство, обозначенное в спецификации `[[Prototype]]`; в нем хранится обычная ссылка на другой объект. Почти у всех объектов в момент их создания этому свойству присваивается значение, отличное от `null`.

Примечание: вскоре вы увидите, что связь `[[Prototype]]` у объекта может быть пустой, хотя такая ситуация встречается относительно редко.

Пример:

```
var myObject = {
    a: 2
};

myObject.a; // 2
```

Для чего нужна ссылка `[[Prototype]]`? В главе 8 была рассмотрена операция `[[Get]]`, которая вызывается при обращении к свойству объекта (например, `myObject.a`). При выполнении операции `[[Get]]` по умолчанию прежде всего проверяется, содержит ли сам объект свойство `a`, и если содержит — то оно используется.

Посредники (proxy) ES6 выходят за рамки обсуждения в этой книге (они будут рассмотрены в одной из следующих книг), но все, что сказано здесь о нормальном поведении `[[Get]]` и `[[Put]]`, не относится к операциям, выполняемым с участием посредников.

Но что происходит, если `a` *отсутствует* в `myObject`? Вот здесь и вступает в игру ссылка `[[Prototype]]` объекта.

Если операция `[[Get]]` по умолчанию не может найти запрашиваемое свойство непосредственно в объекте, она переходит по ссылке `[[Prototype]]` объекта:

```
var anotherObject = {
    a: 2
};

// создать объект, связанный с `anotherObject`
var myObject = Object.create( anotherObject );

myObject.a; // 2
```

Вскоре мы объясним, что делает метод `Object.create(..)` и как он работает. А пока просто считайте, что он создает объект, который связан через `[[Prototype]]` (тема этой главы) с заданным объектом.

Итак, мы создали объект `myObject`, который связан через `[[Prototype]]` с другим объектом. Очевидно, свойство `myObject.a` не существует; тем не менее обращение к свойству завершается успешно (оно обнаруживается в `anotherObject`) и возвращает значение `2`.

Но если бы свойство `a` не было обнаружено и в `anotherObject`, то была бы проверена и его цепочка `[[Prototype]]`, если она не пуста, и поиск перешел бы по ней.

Этот процесс продолжается до тех пор, пока не будет найдено подходящее имя свойства или не завершится цепочка `[[Prototype]]`. Если к концу цепочки подходящее свойство так и не будет найдено, операция `[[Get]]` возвращает `undefined`.

По аналогии с описанным процессом поиска по цепочке `[[Prototype]]`, если вы перебираете содержимое объекта в цикле `for..in`, в *перечисление* будут включены все свойства, достижимые по цепочке (и для которых разрешено перечисление — см. главу 8). Если оператор `in` используется для проверки существования свойства объекта, он проверит всю цепочку объекта (независимо от *перечисляемости*):

```
var anotherObject = {
    a: 2
};

// создать объект, связанный с `anotherObject`
var myObject = Object.create( anotherObject );

for (var k in myObject) {
    console.log("found: " + k);
```

```
}

// found: a
("a" in myObject); // true
```

Итак, цепочка `[[Prototype]]` проверяется по одному звену при выполнении различных операций поиска. Поиск останавливается при обнаружении свойства или завершении цепочки.

Object.prototype

Но *где именно* «заканчивается» цепочка `[[Prototype]]`?

Любая *нормальная* цепочка `[[Prototype]]` завершается на встроенном объекте `Object.prototype`. Этот объект включает разнообразные стандартные возможности, используемые в JS, потому что все нормальные (встроенные, не являющиеся расширениями управляющей среды) объекты JavaScript «происходят» от объекта `Object.prototype` (то есть этот объект находится на вершине их цепочки `[[Prototype]]`).

Среди функциональности, находящейся в этом объекте, можно отметить знакомые вам функции `String()` и `.valueOf()`. В главе 8 была представлена функция `.hasOwnProperty(..)`. Еще одна функция `Object.prototype`, которая вам, возможно, не знакома — `.isPrototypeOf(..)`, — будет рассмотрена позднее в этой главе.

Назначение и замещение свойств

В главе 3 мы упоминали о том, что назначение свойств объекта не сводится к добавлению нового свойства в объект или изменению значения существующего свойства. Вернемся к этой ситуации и рассмотрим ее более подробно:

```
myObject.foo = "bar";
```

Если объект myObject уже содержит нормальное свойство доступа к данным с именем foo (то есть свойство находится непосредственно в объекте), присваивание ограничивается простым изменением значения существующего свойства.

Если foo не присутствует непосредственно в myObject, происходит обход цепочки [[Prototype]], как при операции [[Get]]. Если foo не удается найти нигде в цепочке, свойство foo добавляется прямо в myObject с заданным значением, как и следовало ожидать.

Но если foo уже присутствует где-то выше в цепочке, присваивание myObject.foo = "bar" может привести к более тонкому (и возможно, неожиданному) поведению. Сейчас эта тема будет рассмотрена более подробно.

Если имя свойства foo встречается как в самом объекте myObject, так и на более высоком уровне цепочки [[Prototype]], начинающейся с myObject, это называется *замещением* (shadowing). Свойство foo непосредственно в myObject замещает любое свойство foo, находящееся выше в цепочке, потому что поиск myObject.foo всегда находит свойство foo на самом нижнем уровне цепочки.

Как упоминалось выше, замещение foo в myObject не настолько просто, как может показаться. Рассмотрим три возможные ситуации с присваиванием myObject.foo = "bar", когда foo находится не в myObject, а на более высоком уровне цепочки [[Prototype]] myObject:

1. Если выше в цепочке [[Prototype]] находится нормальное свойство доступа к данным (см. главу 8) с именем foo и оно не помечено как доступное только для чтения (writable:false), то новое свойство с именем foo добавляется прямо в myObject, что приводит к *замещению* существующего свойства.

2. Если свойство foo находится выше в цепочке [[Prototype]] и оно помечено как доступное только для чтения (writable:

`false`), то запрещаются как назначение существующего свойства, так и создание замещенного свойства. Если код выполняется в режиме `strict`, происходит ошибка, а если нет — присваивание значения свойства игнорируется. В любом случае замещение не происходит.

3. Если свойство `foo` находится выше в цепочке `[[Prototype]]` и оно имеет сеттер (см. главу 8), то всегда будет вызываться сеттер. Свойство `foo` не будет добавлено в `myObject` (с замещением), и сеттер `foo` не будет переопределяться.

Многие разработчики считают, что присваивание свойству (`[[Put]]`) всегда приводит к замещению, если свойство уже существует выше в цепочке `[[Prototype]]`, но, как видите, это справедливо только в одной из трех описанных ситуаций (случай 1).

Чтобы заместить `foo` в случаях 2 и 3, вы не сможете использовать присваивание `=`. Вместо этого необходимо использовать `Object.defineProperty(..)` (см. главу 8) для добавления `foo` к `myObject`.

Пожалуй, случай 2 оказывается самым удивительным из трех. Присутствие свойства, доступного только для чтения, блокирует неявное создание (замещение) на более низком уровне цепочки `[[Prototype]]`. Это ограничение объясняется прежде всего целью укрепления иллюзии наследования свойств от классов. Если представить, что `foo` на более высоком уровне цепочки было унаследовано (скопировано) в `myObject`, будет логично подкрепить запрет записи для свойства `foo` в `myObject`. Тем не менее, если отделить иллюзию от факта и понять, что никакого копирования при наследовании на самом деле нет (см. главы 9 и 10), кажется немного странным, что `myObject` запрещается иметь свойство foo только потому, что у другого объекта есть свойство foo, в которое запрещена запись. Еще более странно, что это ограничение распространяется только на присваивание `=`, но не соблюдается при использовании `Object.defineProperty(..)`.

Замещение методов приводит к уродливому явному псевдополиморфизму (см. главу 9). Обычно замещение создает слишком много сложностей и нюансов, поэтому его лучше по возможности избегать. В главе 11 представлен альтернативный паттерн проектирования, который среди прочего стимулирует отказ от замещения в пользу более элегантных решений.

Замещение может происходить неявно; если вы стараетесь избежать его, будьте очень внимательны. Пример:

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject );

anotherObject.a; // 2
myObject.a; // 2

anotherObject.hasOwnProperty( "a" ); // true
myObject.hasOwnProperty( "a" ); // false

myObject.a++; // неявное замещение!

anotherObject.a; // 2
myObject.a; // 3

myObject.hasOwnProperty( "a" ); // true
```

Хотя может показаться, что вызов `myObject.a++` должен (через делегирование) пройти по цепочке и просто увеличить свойство `anotherObject.a` «на месте», вместо этого операция `++` соответствует `myObject.a = myObject.a + 1`. В результате `[[Get]]` ищет свойство по цепочке `[[Prototype]]`, получает текущее значение `2` из `anotherObject.a`, увеличивает значение на 1, после чего `[[Put]]` присваивает значение `3` замещенному свойству `a` в `myObject`. Сюрприз!

Будьте очень внимательны при работе с делегированными свойствами, которые вы изменяете. Если вы хотите увеличить `anotherObject.a`, для этого есть только один правильный способ: `anotherObject.a++`.

«Класс»

Возможно, у вас возник вопрос: почему один объект должен быть связан с другим объектом? Какая от этого польза? Это абсолютно законный вопрос, но чтобы понять, что такое свойство `[[Prototype]]` и какая от него польза, сначала необходимо разобраться, чем `[[Prototype]]` *не является*.

Как объяснялось в главе 9, в JavaScript нет абстрактных «планов» объектов, называемых *классами*, какие существуют в языках, ориентированных на использование классов. В JavaScript есть *только* объекты.

JavaScript занимает почти уникальное положение среди языков: это едва ли не единственный язык, который может с полным правом использовать определение «объектно-ориентированный», потому что он входит в очень короткий список языков, в которых объект может создаваться напрямую, без использования классов.

В JavaScript классы не могут описывать то, что могут делать объекты (потому что они не существуют). Объект сам определяет свое поведение напрямую. Есть *только* объект.

Функции «классов»

В JavaScript есть своеобразный тип поведения, которым в течение многих лет злоупотребляли и пытались сделать его похожим на классы. Этот подход будет рассмотрен подробно.

Это нетипичное поведение «чего-то похожего на классы» основано на странной особенности функций: все функции по умолчанию получают открытое, не перечисляемое (см. главу 8) свойство с именем `prototype`, которое указывает на произвольный объект:

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

Foo.prototype; // { }
```

Этот объект часто называется *прототипом* `Foo`, потому что мы обращаемся к нему по ссылке на свойство `Foo.prototype` (имя которого выбрано не слишком удачно). Тем не менее, как вы вскоре увидите, эта терминология неизбежно приводит к путанице. Вместо этого я буду называть его «объект, прежде называвшийся прототипом `Foo`»... Шутка. Как насчет «объекта, который непонятно почему называется *Foo-точка-прототип*»?

Но что же представляет собой этот объект, как его ни называй?

Самое простое объяснение, что каждый объект, создаваемый вызовом `new Foo()` (см. главу 7), наделяется (отчасти произвольно) связью через `[[Prototype]]` с этим объектом «*Foo-точка-прототип*».

Приведу пример:

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

var a = new Foo();

Object.getPrototypeOf( a ) === Foo.prototype; // true
```

При создании `a` вызовом `new Foo()` среди прочего (все четыре шага описаны в главе 7) он получает внутреннюю ссылку `[[Prototype]]` на объект, на который указывает `Foo.prototype`.

Остановитесь на минутку и поразмыслите над тем, что следует из этого утверждения. В языках, ориентированных на использование классов, можно создать несколько копий (то есть экземпляров) класса — что-то вроде штамповки по готовым формам. Как было показано в главе 9, это происходит из-за того, что процесс создания экземпляров (или наследования от) класса означает «скопировать план поведения из этого класса в физический объект», и это повторяется для каждого нового экземпляра.

Но в JavaScript такое копирование не выполняется. Вы не создаете разные экземпляры класса. Можно создать несколько объектов, связанных с общим объектом через `[[Prototype]]`. Но по умолчанию копирование не происходит, а следовательно, эти объекты не получаются полностью самостоятельными и отсоединенными друг от друга — они остаются связанными между собой.

`new Foo()` приводит к созданию нового объекта (мы назвали его `a`), и этот новый объект `a` во внутренней реализации связывается через `[[Prototype]]` с объектом `Foo.prototype`.

В итоге появляются два объекта, связанных друг с другом. Вот и все. Мы не создавали экземпляр класса. И конечно, поведение не копировалось из «класса» в конкретный объект. Мы просто связали два объекта друг с другом по ссылке.

От многих JS-разработчиков ускользает один секрет: вызов новой функции `Foo()` не имеет практически ничего общего с процессом создания ссылки. Это своего рода побочный эффект. `new Foo()` — непрямой обходной путь для получения нужного результата: нового объекта, связанного с другим объектом.

Можно ли получить этот результат более прямолинейно? Да! На помощь приходит `Object.create(..)`. Но вскоре мы доберемся до него.

Что в имени тебе моем?

В JavaScript вы не копируете поведение из одного объекта («класса») в другой («экземпляр»). Вы создаете связи между объектами. В наглядном представлении механизма `[[Prototype]]` стрелки идут справа налево и снизу вверх (рис. 10.1).

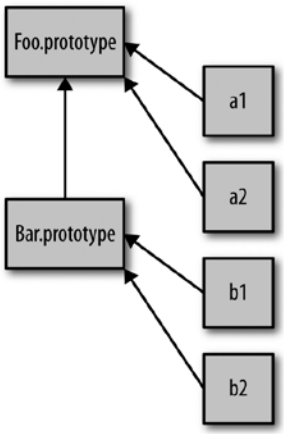

Рис. 10.1

Этот механизм часто называется *наследованием на основе прототипов* (код будет подробно рассмотрен ниже); обычно это называется версией классического наследования для динамических языков. Это попытка воспользоваться общим пониманием того, что означает «наследование» в мире, ориентированном на использование классов, но при этом слегка подстроить (читай: подогнать силой) понятную семантику для динамических сценариев.

Слово «наследование» имеет предельно четкий смысл (см. главу 9) с большой исторической нагрузкой. Простое добавление слов «на основе прототипов», которое должно было отличить наследование от *практически противоположного поведения* в JavaScript, только создало почти два десятилетия невразумительной путаницы.

Я нередко говорю, что добавлять «на основе прототипов» к «наследованию», чтобы радикально изменить реальный смысл термина, это все равно что держать апельсин в одной руке и яблоко в другой и настаивать на том, что яблоко следует называть «красным апельсином». Какой невразумительный ярлык ни налепи, это не изменит того *факта*, что в одной руке у вас яблоко, а в другой — апельсин.

Гораздо лучше и правильнее называть яблоко яблоком — использовать самую точную и прямую терминологию. Это поможет понять как сходство, так и многочисленные различия, потому что у всех нас имеется простое, общее понимание того, что означает «яблоко».

Из-за путаницы и смешения терминов я считаю, что ярлык «наследование на основе прототипов» (и попытки некорректного применения всей сопутствующей терминологии, ориентированной на классы, — «класс», «конструктор», «экземпляр», «полиморфизм» и т. д.) принес больше вреда, чем пользы, для объяснения того, как же *реально* работает механизм JavaScript.

Наследование подразумевает операцию *копирования*, а JavaScript не копирует свойства объектов (по умолчанию). Вместо этого JS создает связь между двумя объектами, благодаря которой один объект может *делегировать* обращения к свойствам/функциям другому объекту. Термин «*делегирование*» (см. главу 11) гораздо точнее описывает механизм связывания объектов в JavaScript.

Другой термин, который тоже иногда приходится слышать в JavaScript, — *дифференциальное наследование*. Идея заключается в том, чтобы поведение объекта описывалось в контексте его *отличий* от более общего дескриптора. Например, можно сказать, что автомобиль — разновидность транспортного средства, которая имеет ровно четыре колеса (вместо того, чтобы описывать всю специфику обобщенного транспортного средства (двигатель и т. д.)).

Если попытаться рассматривать любой конкретный объект в JS как совокупность всего поведения, доступного посредством делегирования, и мысленно «сплющить» все это поведение до одного объекта, вы увидите (более или менее), какое место в этой картине занимает дифференциальное наследование.

Но как и в случае с наследованием на основе прототипов, дифференциальное наследование создает иллюзию того, что модель мышления важнее физически происходящего в языке. Оно упускает из виду тот факт, что объект B не конструируется по разностному принципу, а строится с определением конкретных характеристик — не считая «дыр», в которых не определяется ничего. Именно в этих «дырах» (пропусках в определениях или их отсутствии) делегирование может перехватить управление и на ходу «заполнить» их делегированным поведением.

Стандартная встроенная реализация не создает один «сплющенный» объект посредством копирования, что подразумевает модель мышления дифференциального наследования. Таким образом, дифференциальное наследование не очень естественно подходит для описания того, как работает механизм JavaScript `[[Prototype]]`.

Вы можете использовать терминологию и модель мышления дифференциального наследования — это дело вкуса, но нельзя отрицать тот факт, что они соответствуют только мыслительной акробатике в вашем мозгу, но не физическому поведению в движке.

«Конструкторы»

Вернемся к коду, приведенному выше:

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

var a = new Foo();
```

Что именно заставляет вас думать, что `Foo` является «классом»?

Во-первых, мы видим использование ключевого слова `new`, как в языках, ориентированных на использование классов, при конструировании экземпляров классов. Во-вторых, кажется, что мы действительно выполняем метод-конструктор класса, потому что метод `Foo()` действительно вызывается — подобно тому, как конструктор реального класса вызывается при создании экземпляра этого класса.

Путаница семантики «конструктора» усугубляется тем, что у объекта со странным обозначением `Foo.prototype` в запасе есть еще один трюк. Рассмотрим следующий код:

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

Foo.prototype.constructor === Foo; // true

var a = new Foo();
a.constructor === Foo; // true
```

Объект `Foo.prototype` по умолчанию (в момент объявления в строке 1 фрагмента) получает открытое неперечисляемое (см. главу 8) свойство с именем `.constructor`, и это свойство содержит обратную ссылку на функцию (`Foo` в данном случае), с которой был связан объект. Более того, мы видим, что объект `a`, созданный вызовом «конструктора» `new Foo()`, *вроде бы* тоже содержит свойство `.constructor`, которое аналогичным образом указывает на «функцию, которая создала его».

На самом деле это не так: `a` не содержит свойства `.constructor`, и хотя `a.constructor` действительно дает функцию `Foo`, «конструктор» в данном случае не означает «тот, кто сконструировал», как могло бы показаться. Вскоре мы объясним эту странность.

Да, и еще... По действующим в мире JavaScript соглашениям имя «класса» начинается с прописной буквы, и сам факт использования `Foo` вместо `foo` указывает на то, что мы предполагаем, что это «класс». Ведь это же совершенно очевидно, правильно?!

Эти соглашения настолько сильны, что многие статические анализаторы кода JS предупрежают о вызове new с методом, имя которого начинается со строчной буквы, или если new не вызывается с функцией, начинающейся с прописной буквы. Просто удивительно: чтобы создать видимость «ориентации на использование классов» в JavaScript, мы определяем правила синтаксического анализа кода, которые проверяют, что мы используем прописные буквы, хотя для движка JS прописные буквы ничего не значат!

Конструктор или вызов?

В предыдущем фрагменте возникает соблазн считать `Foo` конструктором, потому что `Foo` вызывается с оператором `new` и вроде бы «конструирует» объект. В действительности `Foo` заслуживает называться «конструктором» ничуть не больше, чем любая другая функция в вашей программе. Сами функции конструкторами *не являются*. Тем не менее, когда вы ставите ключевое слово `new` перед обычным вызовом функции, это делает вызов функции «вызовом конструктора». Можно сказать, что `new` в каком-то смысле захватывает любую нормальную функцию и вызывает ее способом, который конструирует объект (в дополнение к тому, что она еще собирается сделать).

Пример:

```
function NothingSpecial() {
    console.log( "Don't mind me!" );
}

var a = new NothingSpecial();
```

```
// "Don't mind me!"

a; // {}
```

`NothingSpecial` — совершенно обычная функция, но при вызове с `new` она конструирует объект (почти как побочный эффект), который присваивается `a`. Этот вызов был *вызовом конструктора*, но функция `NothingSpecial` сама по себе не является *конструктором*.

Иначе говоря, в JavaScript правильнее всего говорить, что «конструктор» — любая функция, вызываемая с ключевым словом `new`.

Функции не являются конструкторами, но вызовы функций являются «вызовами конструктора» в том и только том случае, если они используются с `new`.

Механика

Можно ли сказать, что это *все* стандартные поводы для пресловутых обсуждений «классов» в JavaScript?

Нет, нельзя. Разработчики JS постарались смоделировать как можно больше функциональности, ориентированной на использование классов:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

var a = new Foo( "a" );
var b = new Foo( "b" );

a.myName(); // "a"
b.myName(); // "b"
```

Этот фрагмент демонстрирует еще два трюка из этой же категории:

1. `this.name = name` добавляет свойство `.name` к каждому объекту (`a` и `b` соответственно; о связывании `this` см. главу 7) по аналогии с тем, как экземпляры классов инкапсулируют значения данных.

2. Пожалуй, `Foo.prototype.myName = ...` выглядит более интересно; эта конструкция добавляет свойство (функцию) в объект `Foo.prototype`. Как ни странно, теперь `a.myName()` работает. Как?

Может показаться, будто в приведенном фрагменте при создании `a` и `b` свойства/функции объекта `Foo.prototype` копируются в каждый из объектов `a` и `b`. Тем не менее этого не происходит.

В начале главы мы объяснили смысл ссылки `[[Prototype]]` и то, как она задействуется в резервном поиске в том случае, если обращение к свойству не удается разрешить в текущем объекте, в контексте алгоритма `[[Get]]` по умолчанию.

Итак, в силу способа их создания каждый из объектов `a` и `b` получает внутреннюю ссылку `[[Prototype]]` на `Foo.prototype`. Когда свойство `myName` не удается найти в `a` и `b` соответственно, оно вместо этого находится (посредством делегирования; см. главу 6) в `Foo.prototype`.

И снова о «конструкторе»

Вспомните, что говорилось ранее о свойстве `.constructor`, и как истинность `a.constructor === Foo` вроде бы означает, что `a` содержит реальное свойство `.constructor`, содержащее ссылку на `Foo`. Нет, это не так.

Это печальное недоразумение. На самом деле ссылка `.constructor` тоже делегируется объекту `Foo.prototype`, который по умолчанию содержит свойство `.constructor`, указывающее на `Foo`.

Вроде бы чрезвычайно удобно, что объект, «сконструированный» `Foo`, имеет доступ к свойству `.constructor`, указывающему на `Foo`. Тем не менее тот факт, что `a.constructor` указывает на `Foo` через делегирование `[[Prototype]]` по умолчанию, — не более чем *совпадение* (почти случайное). Злополучное предположение о том, что `.constructor` означает «тот, кто сконструировал», может подвести вас сразу в нескольких отношениях.

Прежде всего, свойство `.constructor` объекта `Foo.prototype` присутствует по умолчанию только у объекта, созданного при объявлении функции `Foo`. Если вы создадите новый объект и замените ссылку `.prototype` по умолчанию у функции, то новый объект не получит `.constructor` неким волшебным образом.

Пример:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // создание нового объекта
prototype

var a1 = new Foo();
a1.constructor === Foo; // false!
a1.constructor === Object; // true!
```

Ведь объект `a1` не был «сконструирован» `Object(..)`, не так ли? Скорее уж он был «сконструирован» `Foo()`. Многие разработчики считают, что `Foo()` конструирует объект, но если вы думаете, что «конструктор» означает «тот, кто сконструировал», картина тут же распадается — по этой логике свойство `a1.constructor` должно содержать `Foo`, но это не так!

Что происходит? Объект `a1` не содержит свойство `.constructor`, поэтому он делегирует обращение по цепочке `[[Prototype]]` к `Foo.prototype`. Но и этот объект не содержит `.constructor` (в отличие от объекта `Foo.prototype` по умолчанию), поэтому делегиро-

вание продолжается и доходит до `Object.prototype`, вершины цепочки делегирования. А у *этого* объекта имеется свойство `.constructor`, которое указывает на встроенную функцию `Object(..)`.

Неверное представление привело к полному провалу.

Конечно, вы можете добавить `.constructor` в объект `Foo.prototype`, но это придется делать вручную, особенно если вы захотите воспроизвести встроенное поведение и сделать свойство неперечисляемым (см. главу 3).

Пример:

```
function Foo() { /* .. */ }

Foo.prototype = { /* .. */ }; // создание нового объекта
                              // prototype

// Необходимо "исправить" отсутствующее свойство `.constructor`
// нового объекта, заменяющего `Foo.prototype`.
// Описание `defineProperty(..)` см. главу 3.
Object.defineProperty( Foo.prototype, "constructor" , {
    enumerable: false,
    writable: true,
    configurable: true,
    value: Foo    // свойство `.constructor` указывает на `Foo`
} );
```

Слишком много ручной работы для исправления `.constructor`. Более того, фактически мы лишь пытаемся подкрепить неверное представление о том, что «конструктор» означает «тот, кто сконструировал». Иллюзия обходится слишком *дорого*.

По сути свойство `.constructor` объекта указывает по умолчанию на функцию, которая взаимно содержит ссылку на этот объект — ссылку, которая называется `.prototype`. Слова «конструктор» (`.constructor`) и «прототип» (`.prototype`) имеют лишь неформальное значение по умолчанию, которое может оправдаться или не оправдаться в будущем. Лучшее, что вы можете, — почаще напо-

минать себе о том, что «конструктор» не означает «тот, кто сконструировал».

`.constructor` не является каким-то волшебным неизменяемым свойством. Это свойство неперечисляемое (см. предыдущий фрагмент), но оно разрешает запись (то есть может быть изменено). Более того, вы можете добавить или перезаписать (случайно или намеренно) свойство с именем `constructor` у любого объекта в любой цепочке `[[Ptototype]]` любым значением по своему усмотрению.

Из-за правил обхода цепочки `[[Prototype]]` алгоритмом `[[Get]]` ссылка из свойства `.constructor`, где бы она ни находилась, может разрешаться совсем не так, как вы ожидаете.

Теперь видите, как опасно полагаться на ее смысл?

Что же в результате? Не стоит *рассчитывать* на то, что произвольная ссылка из свойств объекта (например, `a1.constructor`) будет считаться предполагаемой ссылкой на некую функцию по умолчанию, как вы того ожидаете. Более того, как вы вскоре увидите, даже если просто не задать нужную информацию, это может кончиться тем, что `a1.constructor` будет указывать на нечто совершенно удивительное и неразумное.

Значение `a1.constructor` в высшей степени ненадежно, и полагаться на него в коде крайне небезопасно. Как правило, таких ссылок следует по возможности избегать.

Наследование (на основе прототипов)

Мы уже видели некоторые приближенные реализации механики классов, которые обычно сооружаются в программах JavaScript.

Однако классы JavaScript были бы довольно бестолковыми, если бы не существовало приближенной реализации «наследования».

На самом деле мы уже видели в действии механизм, обычно называемый «наследованием на основе прототипов», когда объект `a` «наследовал» от `Foo.prototype` и таким образом получал доступ к функции `myName()`. Но мы традиционно думаем о наследовании как о связи между двумя классами, а не между классом и экземпляром (рис. 10.2).

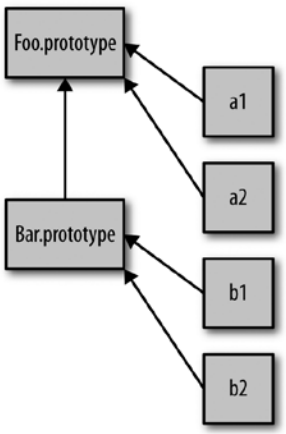

Рис. 10.2

На этой диаграмме, уже приводившейся ранее, обозначено не только делегирование от объекта («экземпляра») `a1` к объекту `Foo.prototype`, но и от `Bar.prototype` к `Foo.prototype`, что отчасти напоминает концепцию наследования классов «родитель-потомок». Конечно, *напоминает*, за исключением направления стрелок, которые показывают, что речь идет о связях посредством делегирования, а не об операциях копирования.

А вот типичный код «в стиле прототипов», который создает такие ссылки:

```
function Foo(name) {
    this.name = name;
}

Foo.prototype.myName = function() {
    return this.name;
};

function Bar(name,label) {
    Foo.call( this, name );
    this.label = label;
}

// здесь мы создаем новый объект `Bar.prototype`,
// связанный с `Foo.prototype`
Bar.prototype = Object.create( Foo.prototype );

// Внимание! Значение `Bar.prototype.constructor` исчезает.
// Возможно, вам придется вручную "исправить" его, если
// вы привыкли полагаться на такие свойства!

Bar.prototype.myLabel = function() {
    return this.label;
};

var a = new Bar( "a", "obj a" );

a.myName(); // "a"
a.myLabel(); // "obj a"
```

Чтобы понять, почему в приведенном фрагменте `this` указывает на `a`, обращайтесь к главе 7.

Самая важная часть — `Bar.prototype = Object.create( Foo.prototype )`. Вызов `Object.create(..)` создает «новый» объект на пустом месте и связывает внутреннее свойство `[[Prototype]]` этого нового объекта с объектом, заданным вами (`Foo.prototype` в данном случае).

Другими словами, эта строка означает: «Создать новый объект `Bar.prototype`, связанный с `Foo.prototype`».

При объявлении функции `Bar() { .. }` `Bar`, как и любая другая функция, содержит ссылку `.prototype` на свой объект по умолчанию. Но *этот* объект не связан с `Foo.prototype`, как бы мы хотели. Поэтому мы создаем новый объект с нужной связью, фактически отбрасывая исходный неправильно связанный объект.

Многие разработчики полагают, будто каждое из двух следующих решений работает. Тем не менее они работают не так, как можно было бы ожидать:

```
// работает не так, как вы ожидали!
Bar.prototype = Foo.prototype;

// работает более или менее так, но с побочными
// эффектами, которые, скорее всего, вам не нужны :(
Bar.prototype = new Foo();
```

Строка `Bar.prototype = Foo.prototype` не создает новый объект для связывания с `Bar.prototype`. Она просто делает `Bar.prototype` еще одной ссылкой на `Foo.prototype`, что фактически связывает `Bar` непосредственно с тем же объектом, с которым связывается `Foo`: `Foo.prototype`. Это означает, что при присваивании (например, `Bar.prototype.myLabel = ...`) вы изменяете не отдельный объект, а общий объект `Foo.prototype`, что повлияет на все объекты, связанные с `Foo.prototype`. Это почти наверняка не то, что вам нужно. Если же это *то*, что вам нужно, то скорее всего, функция `Bar` вам вообще не нужна, используйте только `Foo`, это только упростит код.

`Bar.prototype = new Foo()` действительно создает новый объект, который правильно связан с `Foo.prototype`, как и требовалось. Но для этого он использует «вызов конструктора» `Foo(..)`. Если эта функция имеет какие-либо побочные эффекты (создание записей в журнале, изменение состояния, регистрация у других объектов,

добавление свойств данных к `this` и т. д.), то эти побочные эффекты происходят в момент связывания `this` (и скорее всего, с другим объектом), а не только при создании «потомков» `Bar()`, как вы, вероятно, ожидали.

Итак, остается использовать `Object.create(..)` для создания нового объекта с правильным связыванием, но без побочных эффектов вызова `Foo(..)`. Небольшой недостаток заключается в том, что нам приходится создавать новый объект и терять старый, вместо того чтобы изменить существующий объект по умолчанию, который вы предоставили.

Было бы удобно иметь стандартный и надежный способ изменения связи для существующего объекта. До появления ES6 существовало нестандартное и не полностью кросс-браузерное решение на базе свойства `.__proto__`. В ES6 появилась вспомогательная функция `Object.setPrototypeOf(..)`, которая решает проблему стандартным и предсказуемым образом.

Сравним стандартные методы связывания `Bar.prototype` с `Foo.prototype` до ES6 и в ES6:

```
// до ES6
// теряет существующий объект `Bar.prototype` по умолчанию
Bar.prototype = Object.create( Foo.prototype );

// ES6+
// изменяет существующий объект `Bar.prototype`
Object.setPrototypeOf( Bar.prototype, Foo.prototype );
```

Если забыть о небольшом снижении быстродействия (потере объекта, который позднее уничтожается уборщиком мусора) в решении с `Object.create(..)`, это решение получается более коротким и, возможно, чуть более удобочитаемым, чем решение ES6+. Впрочем, так и или иначе это всего лишь синтаксические «примочки».

Анализ связей «классов»

А если у вас имеется объект (скажем, `a`) и вы хотите определить, какому объекту он делегирует обращения? Анализ экземпляра (просто объекта в JS) на предмет его иерархии наследования (связей делегирования в JS) часто называется *интроспекцией* (а также *отражением* или *рефлексией*) в традиционных объектно-ориентированных средах.

Пример:

```
function Foo() {
    // ...
}
Foo.prototype.blah = ...;
var a = new Foo();
```

Как проанализировать `a` для определения его «происхождения» (связей делегирования)? Первый способ основан на путанице с классами:

```
a instanceof Foo; // true
```

Оператор `instanceof` получает простой объект (левый операнд) и функцию (правый операнд). Он отвечает на вопрос: присутствует ли в цепочке `[[Prototype]]` для `a` объект, на который указывает `Foo.prototype`?

К сожалению, это означает, что вы можете запрашивать информацию о «происхождении» некоторого объекта (`a`), если у вас имеется функция (`Foo` с присоединенной ссылкой `.prototype`) для тестирования. Если у вас есть два произвольных объекта (допустим, `a` и `b`) и вы хотите определить, связаны ли объекты через цепочку `[[Prototype]]`, оператор `instanceof` сам по себе не поможет.

Если вы используете встроенную функцию `.bind(..)` для создания жестко связанной функции (см. главу 7), то у созданной функции не будет свойства .prototype. При использовании `instanceof` с такой функцией происходит прозрачная замена на значение .prototype целевой функции, на основе которой была создана жестко связанная функция.

Применение жестко связанных функций в «вызовах конструкторов» достаточно нетипично. Но если вы их используете, они будут вести себя так, как если бы вместо них использовалась исходная целевая функция, а это означает, что поведение `instanceof` с жестко связанной функцией также соответствует поведению исходной функции.

Следующий фрагмент демонстрирует бессмысленность попыток делать выводы об отношениях между двумя объектами с использованием семантики «классов» и `instanceof`:

```
// вспомогательная функция для проверки того, связан
// ли объект `o1` с `o2` (то есть делегирует обращения)
function isRelatedTo(o1, o2) {
    function F(){}
    F.prototype = o2;
    return o1 instanceof F;
}

var a = {};
var b = Object.create( a );

isRelatedTo( b, a ); // true
```

Внутри `isRelatedTo(..)` мы берем фиктивную функцию `F`, переназначаем ее свойство `.prototype` на некий объект `o2`, после чего спрашиваем, является ли `o1` «экземпляром» `F`. Очевидно, объект `o1` не был потомком функции `F` и даже не конструировался на ее основе, поэтому вам должно быть очевидно, почему подобные

неразумные попытки только создают путаницу. Проблема возникает из-за неуклюжести принудительного моделирования семантики классов в JavaScript, что в данном случае проявляется в косвенной семантике `instanceof`.

Второй, намного более элегантный подход к анализу связей `[[Prototype]]` выглядит так:

```
Foo.prototype.isPrototypeOf( a ); // true
```

Обратите внимание: на этот раз функция `Foo` нам вообще *не нужна*. Нужен только объект (в данном случае произвольно названный `Foo.prototype`), который будет проверяться на родство с другим объектом. Вопрос, на который отвечает `isPrototypeOf(..)`, выглядит так: встречается ли во всей цепочке `[[Prototype]]` объекта `a` объект `Foo.prototype` ?

Тот же вопрос и точно такой же ответ. Но во втором варианте не нужна косвенная ссылка на функцию (`Foo`), у которой будет автоматически проверено свойство `.prototype`.

Нужны только два объекта, которые проверяются на существование косвенной связи. Пример:

```
// Встречается ли b где-то в цепочке
// [[Prototype]] объекта c?
b.isPrototypeOf( c );
```

Обратите внимание: для этого решения функция («класс») вообще не нужна. Оно просто использует ссылки на объекты `b` и `c`, запрашивая информацию об их отношениях. Другими словами, функция `isRelatedTo(..)` встроена в язык, и она называется `isPrototypeOf(..)`.

Также возможно напрямую получить цепочку `[[Prototype]]` объекта. В ES5 стандартным считался следующий способ:

```
Object.getPrototypeOf( a );
```

Заметим, что ссылка на объект будет именно такой, как и ожидалось:

```
Object.getPrototypeOf( a ) === Foo.prototype; // true
```

Многие браузеры (хотя и не все) также давно поддерживают нестандартный альтернативный способ обращения к внутреннему свойству `[[Prototype]]`:

```
a.__proto__ === Foo.prototype; // true
```

Странное свойство `.__proto__` (не стандартизированное до ES6) «волшебным» образом получает внутреннее свойство `[[Prototype]]` объекта в виде ссылки, что достаточно удобно, если вы хотите напрямую проанализировать цепочку (или даже обойти ее: `.__proto__.__proto__...`).

Как было показано ранее для `.constructor`, свойство `.__proto__` не существует фактически в объекте, который вы анализируете (`a` в нашем примере). На самом деле оно существует (как неперечисляемое; см. главу 7) во встроенном объекте `Object.prototype` наряду с некоторыми часто используемыми функциями (`.toString()`, `.isPrototypeOf(..)` и т. д.).

Более того, `.__proto__` выглядит как свойство, но правильнее рассматривать его как геттер/сеттер (см. главу 8).

Реализацию `.__proto__` можно приблизительно представить следующим образом (за определениями свойств объектов обращайтесь к главе 8):

```
Object.defineProperty( Object.prototype, "__proto__", {
    get: function() {
        return Object.getPrototypeOf( this );
    },
    set: function(o) {
        // setPrototypeOf(..) as of ES6
        Object.setPrototypeOf( this, o );
```

```
        return o;
    }
} );
```

Таким образом, когда мы обращаемся (получаем значение) `a.__proto__`, это скорее напоминает вызов `a.__proto__()` (вызов get-функции). У *этого* вызова функции `this` содержит `a`, несмотря на то, что в объекте `Object.prototype` существует get-функция (правила связывания `this` изложены в главе 7), так что вызов фактически эквивалентен `Object.getPrototypeOf( a )`.

Вы также можете изменять `.__proto__`, как при использовании `Object.setPrototypeOf(..)` из ES6 (см. выше). Тем не менее обычно изменять `[[Prototype]]` для существующего объекта не рекомендуется.

В некоторых фреймворках используются очень сложные, нетривиальные средства, которые делают возможными такие трюки, как «создание подклассов» `Array`. Тем не менее в общей практике программирования подобные вещи осуждаются, так как обычно они *серьезно* усложняют понимание/сопровождение кода.

В ES6 ключевое слово `class` также предоставляет средства, позволяющие моделировать «создание подклассов» для встроенных объектов (таких как `Array`). Синтаксис `class`, появившийся в ES6, рассматривается в приложении Г.

Единственным исключением (как упоминалось выше) является случай, когда `[[Prototype]]` объекта по умолчанию `.prototype` функции присваивается ссылка на другой объект (кроме `Object.prototype`). Тем самым предотвращается полная замена объекта по умолчанию новым объектом. В остальных случаях лучше рассматривать связь `[[Prototype]]` объекта как характеристику, доступную только для чтения, — это упростит чтение вашего кода в будущем.

Связи между объектами

Как было показано выше, механизм `[[Prototype]]` представляет собой внутреннюю ссылку в объекте, которая указывает на другой объект.

Эта связь (чаще всего) используется при обращении к свойству/методу первого объекта, если выясняется, что такое свойство/метод в объекте не существует. В таком случае связь `[[Prototype]]` приказывает движку искать свойство/метод в связанном объекте. Если этот объект не может удовлетворить запрос, происходит переход по его ссылке `[[Prototype]]` и т. д. Последовательность связей между объектами образует так называемую «цепочку прототипов».

Создание связей вызовом Create()

Мы подробно разобрались, почему механизм `[[Prototype]]` в JavaScript не является аналогом классов. Как было показано, его суть заключается в создании связей между объектами.

Для чего нужен механизм `[[Prototype]]`? Почему разработчики JS так часто прилагают значительные усилия, пытаясь моделировать классы в своем коде?

Помните, ранее в этой главе говорилось о важной роли `Object.create(..)`? Теперь вы сможете понять, что он делает:

```
var foo = {
    something: function() {
        console.log( "Tell me something good..." );
    }
};

var bar = Object.create( foo );

bar.something(); // Tell me something good...
```

`Object.create(..)` создает новый объект (`bar`), связанный с заданным объектом (`foo`). Таким образом, вы получаете всю мощь механизма `[[Prototype]]` (делегирование), но без функций `new`, моделирующих классы, вызовов конструкторов, непонятных ссылок `.prototype` и `.constructor`, а также прочих ненужных сложностей.

`Object.create(null)` создает объект с пустой (то есть `null`) связью `[[Prototype]]`; такой объект не может ничего делегировать. Поскольку у такого объекта нет цепочки прототипов, оператор `instanceof` (см. выше) не может ничего проверить, и поэтому всегда возвращает `false`. Специальные объекты с пустой связью `[[Prototype]]` часто называются «словарями», так как они обычно используются исключительно для хранения данных в свойствах, прежде всего из-за отсутствия непредвиденных эффектов от делегированных свойств/функций в цепочке `[[Prototype]]`.

Классы *не обязательны* для создания содержательных связей между двумя объектами. Единственное, что должно вас интересовать, — это объекты, связанные между собой для делегирования, а `Object.create(..)` предоставляет необходимые связи без всей шелухи классов.

Полифил Object.create()

Поддержка `Object.create(..)` была добавлена в ES5. Возможно, вам понадобится поддерживать среды, предшествовавшие ES5 (например, старые версии IE), поэтому я приведу простой частичный полифил для `Object.create(..)`, который предоставит эту возможность даже в этих старых средах JS:

```
if (!Object.create) {
    Object.create = function(o) {
```

```
        function F(){}
        F.prototype = o;
        return new F();
    };
}
```

Полифил использует фиктивную функцию `F`; мы переопределяем ее свойство `.prototype` указателем на объект, с которым должна быть установлена связь. Затем конструктор `new F()` используется для создания нового объекта, который имеет заданную связь.

Такое использование `Object.create(..)` является самым распространенным, потому что для него может быть создан полифил. Стандартная встроенная реализация `Object.create(..)` в ES6 предоставляет дополнительные возможности, которые *не могут* моделироваться полифилами в средах, предшествующих ES5. Как следствие, эта функциональность используется значительно реже. Для полноты изложения рассмотрим эту дополнительную функциональность:

```
var anotherObject = {
    a: 2
};

var myObject = Object.create( anotherObject, {
    b: {
        enumerable: false,
        writable: true,
        configurable: false,
        value: 3
    },
    c: {
        enumerable: true,
        writable: false,
        configurable: false,
        value: 4
    }
} );
```

```
myObject.hasOwnProperty( "a" ); // false
myObject.hasOwnProperty( "b" ); // true
myObject.hasOwnProperty( "c" ); // true

myObject.a; // 2
myObject.b; // 3
myObject.c; // 4
```

Второй аргумент `Object.create(..)` задает имена свойств, добавляемых в каждый создаваемый объект, для чего объявляется дескриптор каждого нового свойства (см. главу 8). Поскольку полифилы для дескрипторов свойств до ES5 невозможны, эта расширенная функциональность `Object.create(..)` не может моделироваться полифилами.

Подавляющее большинство применений `Object.create(..)` использует подмножество функциональности, безопасное для полифилов, поэтому разработчики обычно довольствуются частичным полифилом для сред, предшествующих ES5.

Некоторые разработчики выбирают более жесткое представление: они считают, что полифилы не должны определяться для тех функций, которые не могут быть *полностью* смоделированы полифилом. Так как `Object.create(..)` — это одна из функций, для которых возможен лишь частичный полифил, по канонам этой жесткой точки зрения если вам потребуется использовать функциональность `Object.create(..)` в среде до ES5, вместо полифила следует определить собственную функцию и воздержаться от использования имени `Object.create`:

```
function createAndLinkObject(o) {
    function F(){}
    F.prototype = o;
    return new F();
}

var anotherObject = {
```

```
    a: 2
};

var myObject = createAndLinkObject( anotherObject );

myObject.a; // 2
```

Я не разделяю столь жесткого мнения и полностью поддерживаю распространенный частичный полифил `Object.create(..)`, приведенный выше, и использование его в коде для сред, предшествующих ES5. Принимайте решение сами.

Связи как резерв?

Заманчиво представлять, что связи между объектами прежде всего предоставляют некий резервный механизм для поиска «отсутствующих» свойств и методов. Да, этот результат наблюдается на поверхности, но я не думаю, что он представляет правильное отношение к `[[Prototype]]`.

Пример:

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.cool(); // "cool!"
```

Этот код работает благодаря `[[Prototype]]`. Но если вы напишете его так для того, чтобы объект `anotherObject` действовал в качестве резерва на случай, если `myObject` не может предоставить некоторое свойство/метод, запрашиваемое программой, скорее всего, это только усложнит понимание и сопровождение кода.

Безусловно, ситуации, для которых создавался паттерн проектирования «резерв», существуют, тем не менее для JS они не типичны и не идиоматичны. Если вы видите, что вам приходится применять это решение, возможно, стоит сделать шаг назад и поразмыслить над тем, насколько разумно спроектирована ваша программа.

В ES6 появилась расширенная функциональность Proxy, которая может предоставить некое подобие поведения для ситуаций «метод не найден». Тема Proxy выходит за рамки книги, но будет подробно рассмотрена в одной из последующих книг этой серии.

Не упустите один важный, но неочевидный момент.

Если вы проектируете свой продукт так, чтобы, скажем, при вызове `myObject.cool()` некая работа выполнялась даже в том случае, если в `myObject` не определен метод `cool()`, вы тем самым вводите в API «волшебство» и преподносите сюрприз тем разработчикам, которые будут заниматься сопровождением вашего продукта.

Однако API можно спроектировать так, чтобы он был менее «волшебным», но при этом использовал всю мощь связывания `[[Prototype]]`:

```
var anotherObject = {
    cool: function() {
        console.log( "cool!" );
    }
};

var myObject = Object.create( anotherObject );

myObject.doCool = function() {
    this.cool(); // внутреннее делегирование!
};

myObject.doCool(); // "cool!"
```

Здесь вызывается `myObject.doCool()` — метод, который действительно существует в `myObject`, что делает структуру API более явной (менее «волшебной»). При этом внутренняя реализация следует паттерну делегирования (см. главу 6) и использует делегирование `[[Prototype]]` для передачи обращения `anotherObject.cool()`.

Иначе говоря, делегирование обычно создает меньше путаницы, если оно является подробностью внутренней реализации, а не явно выражается в проектировании API. Делегирование более подробно рассмотрено в следующей главе.

Итоги

При попытке обращения к свойству для объекта, у которого это свойство отсутствует, внутренняя связь `[[Prototype]]` объекта определяет, где операция `[[Get]]` (см. главу 8) должна продолжить поиск. Система каскадных ссылок между объектами фактически определяет «цепочку прототипов» (нечто вроде иерархической цепочки областей видимости) объектов, которые должны перебираться при разрешении свойств.

У всех нормальных объектов на вершине цепочки прототипов находится встроенный объект `Object.prototype` (аналог глобальной области видимости при поиске в областях видимости), на котором останавливается разрешение свойств, не найденных в предыдущих звеньях цепочки. Объект `Object.prototype` содержит `toString()`, `valueOf()` и ряд других общих функций; это объясняет, почему они доступны для всех объектов в языке.

Самый распространенный способ связывания двух объектов — использование ключевого слова `new` с вызовом функции. В результате во время четырех фаз выполнения (см. главу 9) создается новый объект, связанный с другим объектом.

«Другим объектом», с которым связывается новый объект, становится объект, на который указывает свойство с (неудачным) именем `.prototype` функции, вызываемой с `new`. Функции, вызываемые с `new`, часто называются «конструкторами», хотя они и не пытаются создавать экземпляры классов, как это делают *конструкторы* в традиционных языках, ориентированных на использование классов.

Хотя эти механизмы JavaScript на первый взгляд напоминают «создание экземпляров» и «наследование классов» из традиционных языков, ориентированных на использование классов, ключевое различие заключается в том, что в JavaScript копирование не производится. Вместо этого объекты связываются друг с другом по внутренней цепочке `[[Prototype]]`.

По разнообразным причинам, в том числе из-за распространенной терминологии, «наследование» (и «наследование на основе прототипов»), а также все остальные ОО-термины просто не подходят для описания того, как реально работают механизмы JavaScript (а не только применительно к нашим моделям мышления).

Правильнее здесь было бы использовать термин «делегирование», потому что отношения представляются не *копиями*, а связями делегирования.

11 Делегирование поведения

В главе 10 мы довольно подробно рассмотрели механизм `[[Prototype]]` и выяснили, почему было бы неправильно и неуместно описывать его в контексте «классов» или «наследования» (несмотря на все бесчисленные попытки на протяжении почти двух десятилетий). Мы разобрали не только довольно громоздкий синтаксис (`.prototype`, загромождающие код), но и различные ловушки (например, странную систему разрешения `.constructor` или уродливый псевдополиморфный синтаксис). Также были исследованы вариации на тему «примесей», которые применяются многими разработчиками в попытках сгладить существующие неровности.

У многих разработчиков к этому моменту возникает вопрос: почему для того, чтобы сделать что-то настолько простое, вы сталкиваетесь с такими сложностями? Теперь, когда мы откинули занавес и увидели, как некрасиво все устроено, не приходится удивляться тому, что многие разработчики JS никогда не заглядывают на такие глубины, а вместо этого делегируют все проблемы библиотеке «классов», которая делает все за них.

Надеюсь, к этому моменту вы уже не захотите пройти мимо и оставить все подробности библиотеке — «черному ящику». Теперь посмотрим, как можно и нужно относиться к механизму объектов `[[Prototype]]` в JS и почему такое отношение *намного проще и бесхитростнее* всей этой путаницы с классами.

Кратко напомню наши выводы из главы 10: механизм `[[Prototype]]` основан на внутренней ссылке, которая существует в одном объекте и указывает на другой объект.

Эта связь используется в том случае, если программа обращается к свойству/методу первого объекта, но такое свойство/метод не существует. В этом случае связь `[[Prototype]]` сообщает движку, что поиск свойства/метода следует продолжить в связанном объекте. В свою очередь, если этот объект не может удовлетворить запрос, происходит переход по его ссылке `[[Prototype]]` и т. д. Эта серия ссылок между объектами образует так называемую «цепочку прототипов».

Иначе говоря, вся суть механизма, все то, что важно для функциональности, которая может использоваться в JavaScript, **заключается в связях одних объектов с другими объектами**.

Это чрезвычайно фундаментальное наблюдение исключительно важно для понимания мотивации и решений, описанных в этой главе!

Проектирование, ориентированное на делегирование

Чтобы понять, как проще и прямолинейнее использовать механизм `[[Prototype]]`, необходимо понять, что он представляет паттерн проектирования, принципиально отличающийся от паттерна «класс» (см. главу 9).

Некоторые принципы проектирования, ориентированного на использование классов, не теряют своей актуальности, так что не отбрасывайте все, что знаете (только большую часть). Например, инкапсуляция — достаточно мощный механизм, который совместим с делегированием (хотя используется не так часто).

Вы должны постараться поменять стиль мышления и переключиться с паттерна «класс/наследование» на паттерн «делегирование поведения». Если программирование, которым вы занимались во время учебы/работы, было в основном ориентировано на использование классов, новый подход может показаться неудобным или неестественным. Возможно, вам придется несколько раз повторить это умственное упражнение, чтобы привыкнуть к сильно отличающемуся стилю мышления.

Сначала мы рассмотрим несколько теоретических упражнений, а затем перейдем к более конкретному примеру, чтобы у вас сформировался практический контекст для написания собственного кода.

Теория классов

Предположим, есть несколько похожих задач («XYZ», «ABC» и т. д.), которые должны моделироваться в вашем коде.

С классами архитектура может выглядеть примерно так: вы определяете общий родительский (базовый) класс, например `Task`. Этот класс содержит общее поведение для всех «похожих» задач. Затем вы определяете дочерние классы `XYZ` и `ABC`, наследующие от `Task`; каждый класс добавляет специализированное поведение для своей задачи.

Что еще важнее, паттерн проектирования «класс» поощряет использование переопределения методов (и полиморфизма), когда

вы переопределяете обобщенные методы Task в своей задаче XYZ — возможно, даже с использованием super для вызова базовой версии метода с добавлением нового поведения. Вероятно, вы найдете в своем коде немало мест, в которых общее поведение можно было бы «абстрагировать» в родительском классе, чтобы специализировать (переопределить) его в дочерних классах.

Приблизительный псевдокод для такой ситуации:

```
class Task {
    id;

    // конструктор `Task()`
    Task(ID) { id = ID; }
    outputTask() { output( id ); }
}

class XYZ inherits Task {
    label;

    // конструктор `XYZ()`
    XYZ(ID,Label) { super( ID ); label = Label; }
    outputTask() { super(); output( label ); }
}

class ABC inherits Task {
    // ...
}
```

Теперь вы можете создать несколько экземпляров дочернего класса XYZ и использовать эти экземпляры для выполнения задачи «XYZ». Эти экземпляры содержат копии как общего поведения, определенного в Task, так и специализированного поведения, определенного в XYZ. Аналогичным образом экземпляры класса ABC содержат копии поведения Task и специализированного поведения ABC. После конструирования вы обычно взаимодействуете только с этими экземплярами (а не с классами), так как каждый

экземпляр содержит копии всего поведения, необходимого для выполнения своей задачи.

Теория делегирования

А теперь попробуем взглянуть на ту же предметную область, используя делегирование поведения вместо классов.

Сначала вы определяете объект (не класс и не функцию, что бы ни считали многие разработчики JS) с именем `Task`. Он содержит конкретное поведение со вспомогательными методами, которые могут использоваться различными задачами (то есть по сути *делегировать* поведение). Затем для каждой задачи («XYZ», «ABC») определяется объект, в котором хранятся все данные/поведение этой конкретной задачи. Объект конкретной задачи *связывается* с объектом `Task` и может делегировать ему обращения, когда потребуется.

На происходящее можно взглянуть так: для выполнения задачи «XYZ» требуется поведение двух одноранговых объектов (`XYZ` и `Task`). Но вместо того чтобы объединять их посредством копирования из классов, вы храните эти аспекты поведения в разных объектах, а объект `XYZ` может делегировать обращения `Task` по мере необходимости.

Простой код показывает, как это может происходить:

```
Task = {
    setID: function(ID) { this.id = ID; },
    outputID: function() { console.log( this.id ); }
};

// `XYZ` делегирует обращения `Task`
XYZ = Object.create( Task );

XYZ.prepareTask = function(ID,Label) {
```

```
    this.setID( ID );
    this.label = Label;
};

XYZ.outputTaskDetails = function() {
    this.outputID();
    console.log( this.label );
};

// ABC = Object.create( Task );
// ABC ... = ...
```

В этом коде `Task` и `XYZ` не являются классами (или функциями) — это просто объекты. `XYZ` вызовом `Object.create(..)` настраивается для `[[Prototype]]`-делегирования объекту `Task` (см. главу 10).

В отличие от программирования, ориентированного на использование классов (то есть объектно-ориентированного), **я называю такой стиль программирования OLOO** (Objects Linked to Other Objects, то есть «объекты, связанные с другими объектами»). *На самом деле* нас интересует лишь то, что объект `XYZ` делегирует обращения объекту `Task` (как и объект `ABC`).

В JavaScript механизм `[[Prototype]]` связывает объекты с другими объектами. В этом языке нет абстрактных механизмов вроде «классов», как бы вы ни пытались убедить себя в обратном. Следовать по этому пути — все равно что грести на каноэ вверх по реке: в принципе *возможно*, но вы *сознательно* идете против естественного течения. Разумеется, вам будет труднее добраться туда, куда вы направляетесь.

Еще несколько отличий, на которые стоит обратить внимание в OLOO-коде:

1. Поля данных `id` и `label` в приведенном примере представляют собой свойства данных, определенные непосредственно в `XYZ` (но не в `Task`). В общем случае при делегировании `[[Prototype]]`

состояние желательно хранить в делегаторах (`XYZ`, `ABC`), но не в делегате (`Task`).

2. С паттерном проектирования «класс» мы намеренно используем имя `outputTask` как в родителе (`Task`), так и в потомке (`XYZ`), чтобы использовать переопределение (полиморфизм). При делегировании поведения происходит обратное: мы всеми способами стараемся предотвратить совпадение имен на разных уровнях цепочки `[[Prototype]]` (замещение — см. главу 10), потому что конфликты имен создают неуклюжий/ненадежный синтаксис для устранения неоднозначности ссылок (см. главу 9), а этого следует по возможности избегать.

   С этим паттерном стоит отказаться от обобщенных имен методов, которые могут случайно переопределяться. Вместо них следует выбирать содержательные имена, ассоциированные с конкретным типом поведения каждого объекта. Такая схема выбора имен упрощает понимание/сопровождение кода, потому что имена методов (не только в точке определения, но и во всем коде) становятся более понятными (самодокументируемыми).

3. `this.setID(ID);` внутри метода объекта `XYZ` сначала ищет `setID(..)` в `XYZ`, но поскольку метод с таким именем не удается найти в `XYZ`, вследствие *делегирования* `[[Prototype]]` заставляет перейти по ссылке к объекту `Task`. Конечно, здесь этот метод успешно находится. Более того, из-за правил неявного связывания на месте вызова (см. главу 7) при выполнении `setID(..)` даже при том, что метод был найден в `Task`, связывание `this` для этого вызова функции даст `XYZ` — то, что нужно и чего следовало ожидать. То же самое происходит позднее в листинге с `this.outputID()`.

4. Другими словами, обобщенные вспомогательные методы, существующие в `Task`, доступны для разработчика, когда он

взаимодействует с XYZ, поскольку XYZ может делегировать обращения Task.

Делегирование поведения означает, что вы разрешаете некоторому объекту (XYZ) предоставлять делегирование (к Task) для обращений к свойствам и методам, не найденным в объекте (XYZ).

Это *чрезвычайно мощный* паттерн проектирования, сильно отличающийся от представлений о родительских и дочерних классах, наследовании, полиморфизме и т. д. Вместо того чтобы мысленно выстраивать объекты по вертикали (от верхних родителей к нижним потомкам), представьте себе объекты как расположенные рядом друг с другом на одном уровне; связи делегирования проходят между объектами в любом направлении так, как вам потребуется.

Делегирование лучше использовать как подробность внутренней реализации, чем предоставлять напрямую при проектировании API. В предыдущем примере мы не обязательно требуем, чтобы разработчики вызывали XYZ.setID() (хотя это возможно, конечно). Можно сказать, что делегирование скрывается как внутренняя подробность API, когда XYZ.prepareTask(..) делегирует вызов Task.setID(..). За подробностями обращайтесь к разделу «Связи как резерв?» главы 10.

Взаимное делегирование (запрещено)

Нельзя создать цикл, в котором два и более объекта осуществляют взаимное делегирование (двустороннее) относительно друг друга. Если вы связываете B с A, а потом пытаетесь связать A с B, произойдет ошибка.

Печально, что взаимное делегирование запрещено (не особо удивительно, но малость раздражает). Если вы обратитесь к свойству/

методу, которые не существуют в одном из мест, возникнет бесконечная рекурсия в цикле `[[Prototype]]`. Но если все ссылки присутствуют на своих местах, то B может делегировать A и наоборот, и такое решение *может* работать. Таким образом, любой объект сможет делегировать обращения другому объекту для различных целей. В некоторых узкоспециализированных ситуациях это может быть полезно.

Тем не менее взаимное делегирование запрещено, потому что разработчики движка заметили, что эффективнее проверять циклические ссылки (и запрещать их) во время присваивания, чем терять время на защитную проверку при каждом обращении к свойству объекта.

Отладка

Сейчас мы кратко рассмотрим одну тонкость, которая может сбить с толку разработчиков. В общем случае спецификация JS не управляет тем, как инструменты разработчика в браузере должны представлять конкретные значения/структуры разработчику, так что каждый браузер/движок может свободно интерпретировать их так, как считает нужным. Конечно, между браузерами/инструментами порой возникают расхождения. А если говорить конкретно, то поведение, которое мы сейчас изучаем, в настоящее время наблюдается только в инструментарии разработчика Chrome.

Возьмем традиционный код JS в стиле «конструктора класса» так, как он будет выглядеть на консоли инструментов разработчика Chrome:

```
function Foo() {}

var a1 = new Foo();

a1; // Foo {}
```

Взгляните на последнюю строку этого фрагмента: результат вычисления выражения `a1`, для которого выводится `Foo {}`. Если вы попытаетесь выполнить тот же код в Firefox, то, скорее всего, увидите `Object {}`. Почему они отличаются? И что означают эти результаты?

Chrome по сути говорит «`{}` — пустой объект, который был сконструирован функцией с именем `Foo`». Firefox говорит «`{}` — пустой объект, сконструированный на базе `Object`». Тонкое различие заключается в том, что Chrome активно отслеживает в своем *внутреннем свойстве* имя реальной функции, которая использовалась при конструировании, тогда как другие браузеры эту дополнительную информацию не сохраняют.

Было бы соблазнительно попытаться объяснить это механизмами JavaScript:

```
function Foo() {}

var a1 = new Foo();

a1.constructor; // Foo(){}
a1.constructor.name; // "Foo"
```

Значит, вот как Chrome выводит `Foo` — просто анализируя свойство `constructor.name` объекта? Как ни странно, и да, и нет.

Возьмем следующий код:

```
function Foo() {}

var a1 = new Foo();

Foo.prototype.constructor = function Gotcha(){};

a1.constructor; // Gotcha(){}
a1.constructor.name; // "Gotcha"

a1; // Foo {}
```

Хотя мы изменяем значение `a1.constructor.name`, чтобы свойство содержало что-то другое (`Gotcha`), на консоли Chrome все равно используется имя `Foo`.

Похоже, ответ на предыдущий вопрос (используется ли значение `.constructor.name`?) — «нет»; значение должно отслеживаться где-то в другом месте, во внутренних данных.

Но не торопитесь! Посмотрим, как такое поведение работает с кодом в OLOO-стиле:

```
var Foo = {};

var a1 = Object.create( Foo );

a1; // Object {}

Object.defineProperty( Foo, "constructor", {
    enumerable: false,
    value: function Gotcha(){}
});

a1; // Gotcha {}
```

Ага! На этот раз консоль Chrome *нашла* и использовала `.constructor.name`. На самом деле во время написания книги это конкретное поведение было описано как ошибка в Chrome, а к тому моменту, когда вы будете ее читать, ошибка уже может быть исправлена. Возможно, вы увидите исправленный результат `a1; // Object {}`.

Кроме этой ошибки, внутреннее отслеживание (очевидно, предназначенное только для целей отладочного вывода) «имени конструктора» в Chrome — сознательное расширение поведения за пределы требований спецификации JS, реализованное только в Chrome.

Если вы не используете «конструктор» для создания своих объектов (чего мы не рекомендовали делать с OLOO-кодом в этой главе), то вы получите объекты, для которых Chrome *не* отслеживает внутреннее «имя конструктора». Такие объекты будут правильно выводиться в виде `Object {}`, что означает «объект, созданный на основе конструирования `Object()`».

Не стоит думать, что здесь проявляется какой-то недостаток программирования в стиле OLOO. Когда вы программируете в OLOO-стиле и применяете делегирование поведения как паттерн проектирования, вопрос о том, кто «сконструировал» объект (то есть *какая функция* была вызвана с `new`), становится несущественной подробностью. Отслеживание конкретного внутреннего «имени конструктора» в Chrome принесет пользу только в том случае, если вы в полной мере принимаете программирование в стиле классов, но бесполезно, если вы вместо этого принимаете делегирование OLOO.

Сравнение моделей мышления

Теперь, когда вы понимаете различия между паттернами проектирования «класс» и «делегирование» (хотя бы теоретически), давайте рассмотрим последствия этих паттернов для моделей мышления, которые мы используем в своих рассуждениях о коде.

Мы разберем еще несколько примеров теоретического кода (`Foo`, `Bar`) и сравним оба варианта (OO против OLOO) реализации кода. В первом фрагменте используется классический OO-стиль («прототипический»):

```
function Foo(who) {
    this.me = who;
}
Foo.prototype.identify = function() {
    return "I am " + this.me;
```

```
};

function Bar(who) {
    Foo.call( this, who );
}
Bar.prototype = Object.create( Foo.prototype );

Bar.prototype.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};

var b1 = new Bar( "b1" );
var b2 = new Bar( "b2" );

b1.speak();
b2.speak();
```

От родительского класса `Foo` наследует дочерний класс `Bar`, для которого затем создаются два экземпляра — `b1` и `b2`. Здесь `b1` делегирует `Bar.prototype`, который делегирует `Foo.prototype`. К этому моменту происходящее должно выглядеть довольно знакомо. Здесь нет ничего принципиально нового.

А теперь реализуем *абсолютно ту же функциональность* кодом в стиле OLOO:

```
Foo = {
    init: function(who) {
        this.me = who;
    },
    identify: function() {
        return "I am " + this.me;
    }
};

Bar = Object.create( Foo );

Bar.speak = function() {
    alert( "Hello, " + this.identify() + "." );
};
```

```
var b1 = Object.create( Bar );
b1.init( "b1" );
var b2 = Object.create( Bar );
b2.init( "b2" );

b1.speak();
b2.speak();
```

Мы пользуемся теми же преимуществами делегирования `[[Prototype]]` от `b1` к `Bar` и `Foo`, как и в предыдущем фрагменте между `b1`, `Bar.prototype` и `Foo.prototype`. *Получаем все те же три объекта, связанных друг с другом.*

Но что гораздо важнее, все происходящее сильно упрощается, потому что на этот раз мы просто настраиваем связи между объектами без всей запутанной шелухи, которая пытается походить на классы (но не обладает их поведением) с конструкторами, прототипами и вызовами `new`.

Спросите себя: если я могу получить с кодом в стиле OLOO ту же функциональность, что и с кодом в стиле классов, но код OLOO проще и не заставляет помнить о многочисленных подробностях, разве OLOO не лучше?

Проанализируем модели мышления, задействованные в двух фрагментах. Прежде всего фрагмент кода в стиле классов подразумевает модель с сущностями и отношениями между ними (рис. 11.1).

Вообще говоря, это немного нечестно — ведь на этой диаграмме показано много лишних подробностей, которые *на техническом уровне* не нужно знать постоянно (хотя и нужно понимать их). Один из выводов, которые можно сделать, что система связей получается довольно сложной. Но есть и другой вывод: если вы не пожалеете времени на то, чтобы пройти по этим стрелкам, в механизмах JS проявляется внутренняя целостность.

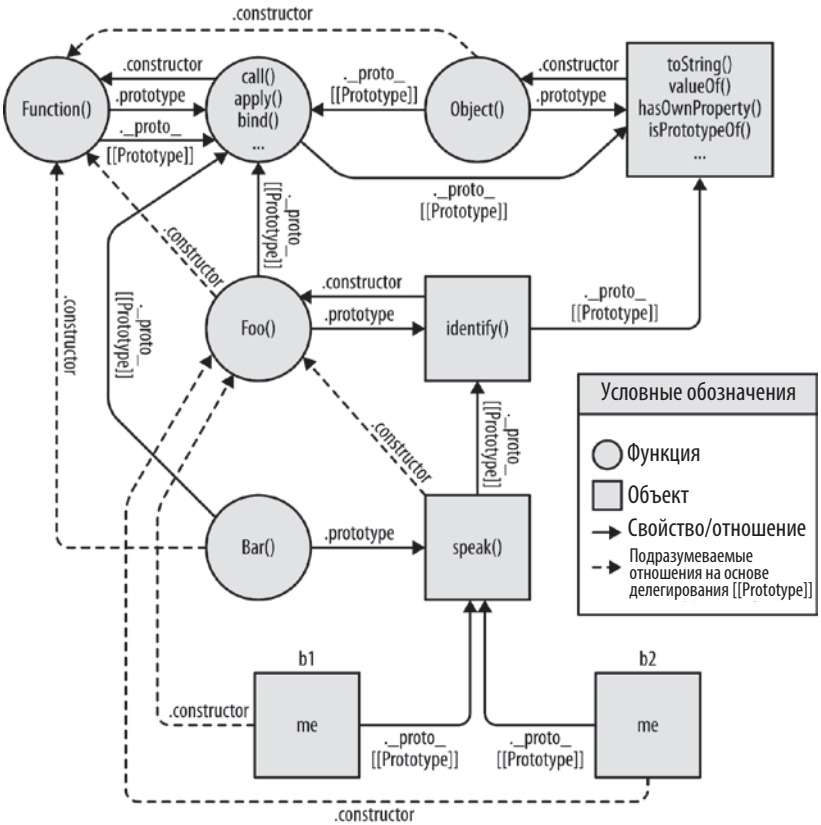

Рис. 11.1

Например, способность функций JS обращаться к `call(..)`, `apply(..)` и `bind(..)` (см. главу 7) объясняется тем, что сами функции являются объектами, а у объектов функций также имеется связь `[[Prototype]]` с объектом `Function.prototype`, определяющим эти методы по умолчанию, которым может делегировать любой объект функции. JS это может, и *вы тоже можете*!

А теперь рассмотрим *слегка* упрощенную версию этой диаграммы, чуть более «честную» для сравнения — на ней показаны только *важные* сущности и связи (рис. 11.2).

Все равно сложно? Пунктирные линии обозначают подразумеваемые отношения при создании «наследования» между `Foo.prototype` и `Bar.prototype`, если вы еще не исправили отсутствующую

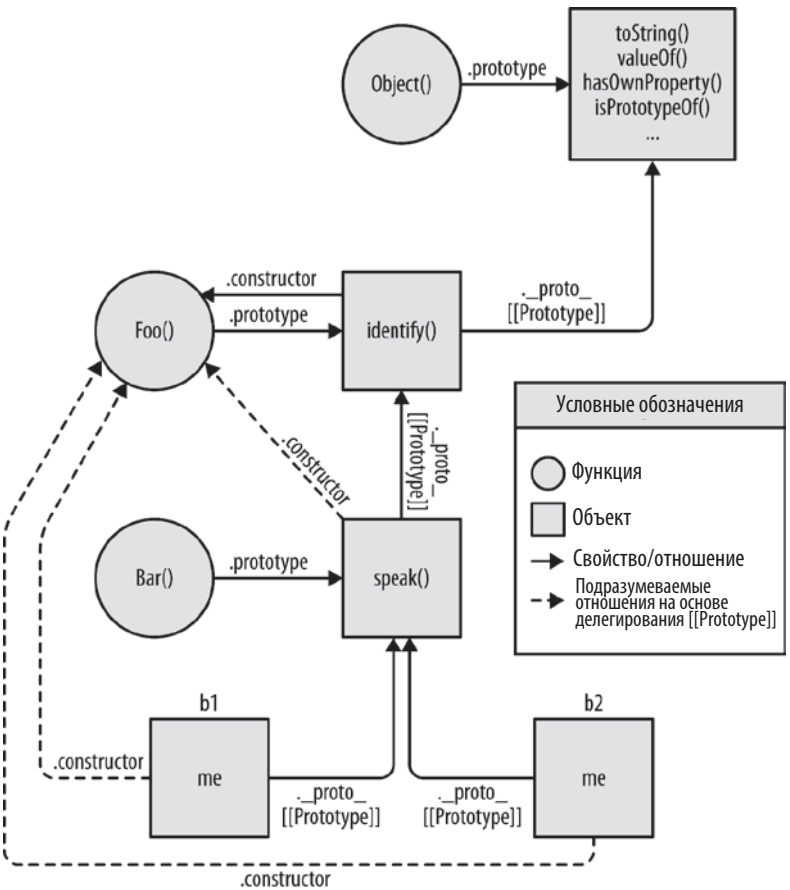

Рис. 11.2

ссылку на свойство `.constructor` (см. раздел «И снова о “конструкторе”» главы 10). Даже при удалении пунктирных линий модель все еще остается слишком сложной, чтобы припоминать ее каждый раз, когда вы используете связи между объектами.

Теперь рассмотрим мысленную модель для кода в стиле OLOO (рис. 11.3).

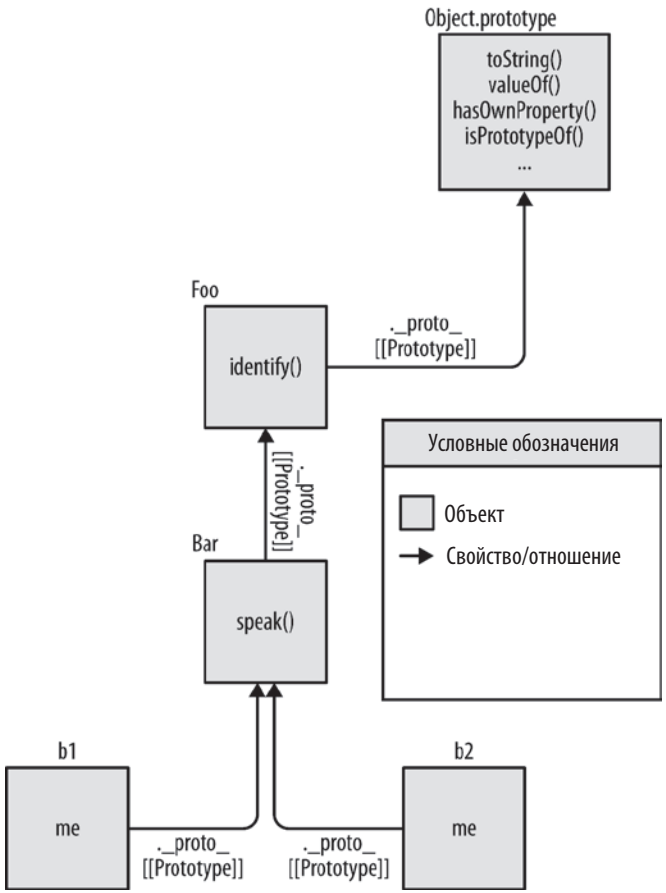

Рис. 11.3

При сравнении достаточно очевидно, что в коде в стиле OLOO гораздо меньше причин для беспокойства, потому что этот код принимает тот факт, что нас интересует только одно: *объекты, связанные с другими объектами*.

Вся шелуха «классов» была лишь запутанным и сложным способом получения того же конечного результата. Откажитесь от нее, и ситуация упростится (без потери каких-либо возможностей).

Классы и объекты

Мы рассмотрели различные теоретические исследования и сравнения моделей мышления «классов» и «делегирования поведения». Но давайте рассмотрим более конкретные сценарии, чтобы вы поняли, как использовать эти концепции.

Начнем с типичного сценария для разработчиков интерфейсной части: создания UI-виджетов (кнопок, раскрывающихся списков и т. д.).

«Классы» виджетов

Вероятно, вы все еще привыкли к паттернам объектно-ориентированного проектирования и поэтому немедленно будете рассматривать предметную область задачи в контексте родительского класса (например, `Widget`), содержащего общее базовое поведение виджетов, и производных дочерних классов для конкретных разновидностей виджетов (например, `Button`).

Для манипуляций с DOM и CSS будет использоваться jQuery — это всего лишь техническая подробность, которая не имеет особого значения для целей текущего обсуждения. Для этого кода абсолютно несущественно, какой фреймворк (jQuery,

Dojo, YUI и т. д.) используется для этих рутинных операций (и используется ли вообще).

Теперь посмотрим, как бы вы реализовали классическую архитектуру «классов» на чистом JS без каких-либо вспомогательных библиотек или синтаксиса «классов»:

```
// Родительский класс
function Widget(width,height) {
    this.width = width || 50;
    this.height = height || 50;
    this.$elem = null;
}

Widget.prototype.render = function($where){
    if (this.$elem) {
        this.$elem.css( {
            width: this.width + "px",
            height: this.height + "px"
        } ).appendTo( $where );
    }
};

// дочерний класс
function Button(width,height,label) {
    // "super" constructor call
    Widget.call( this, width, height );
    this.label = label || "Default";
    this.$elem = $( "<button>" ).text( this.label );
}

// Заставить `Button` "наследовать" от `Widget`
Button.prototype = Object.create( Widget.prototype );
// переопределить базовую "унаследованную" версию `render(..)`
Button.prototype.render = function($where) {
    // вызов "super"
    Widget.prototype.render.call( this, $where );
    this.$elem.click( this.onClick.bind( this ) );
};
```

```
Button.prototype.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

ОО-паттерны проектирования предлагают объявить базовую версию `render(..)` в родительском классе, а затем переопределить ее в дочернем классе, не для того, чтобы заменить ее, а скорее чтобы дополнить базовую функциональность поведением, специфическим для кнопок.

Обратите внимание на уродство явного псевдополиморфизма (см. главу 9) со ссылками `Widget.call` и `Widget.prototype.render.call` для имитации вызовов базовых методов «родительского класса из методов «дочернего класса».

Классы в ES6

Синтаксические удобства для работы с классами ES6 подробно рассмотрены в приложении Г. Но пока кратко продемонстрируем, как реализовать тот же код с использованием классов:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
```

```
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button ‘" + this.label + "’ clicked!" );
    }
}

$( document ).ready( function(){
    var $body = $( document.body );
    var btn1 = new Button( 125, 30, "Hello" );
    var btn2 = new Button( 150, 40, "World" );

    btn1.render( $body );
    btn2.render( $body );
} );
```

Несомненно, синтаксис `class` в ES6 сглаживает многие синтаксические уродства классического подхода. Наличие `super(..)` на первый взгляд кажется удобным (хотя если присмотреться повнимательнее, здесь тоже не все гладко).

Несмотря на улучшения синтаксиса, это не настоящие классы, поскольку они все еще работают на основе механизма `[[Prototype]]`.

Они страдают от тех же несоответствий моделей мышления, которые были исследованы в главах 10 и 11 и до настоящего момента в этой главе. В приложении А подробно рассматривается синтаксис ES6 и его следствия. Вы увидите, почему преодоление синтаксических колдобин не избавляет от путаницы с классами в JS, хотя новый синтаксис доблестно старается выдать себя за полноценное решение!

Какой бы вариант вы ни выбрали — классический синтаксис с прототипами или новые синтаксические удобства ES6, вы все равно *принимаете решение* о моделировании предметной области задачи (UI-виджетов) «классами». И как я постарался показать в последних главах, в JavaScript это решение оборачивается лишними проблемами и бременем следования модели.

Делегирование для объектов Widget

Простой пример `Widget/Button` с использованием делегирования в стиле OLOO:

```
var Widget = {
    init: function(width,height){
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    },
    insert: function($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
};
```

```
var Button = Object.create( Widget );

Button.setup = function(width,height,label){
    // делегированный вызов
    this.init( width, height );
    this.label = label || "Default";
    this.$elem = $( "<button>" ).text( this.label );
};

Button.build = function($where) {
    // делегированный вызов
    this.insert( $where );
    this.$elem.click( this.onClick.bind( this ) );
};

Button.onClick = function(evt) {
    console.log( "Button '" + this.label + "' clicked!" );
};

$( document ).ready( function(){
    var $body = $( document.body );

    var btn1 = Object.create( Button );
    btn1.setup( 125, 30, "Hello" );

    var btn2 = Object.create( Button );
    btn2.setup( 150, 40, "World" );

    btn1.build( $body );
    btn2.build( $body );
} );
```

В этом решении в стиле OLOO `Widget` мы не рассматриваем как родителя или `Button` как потомка. `Widget` — просто объект, своего рода подборка разной функциональности, которому можно делегировать любой конкретный тип виджета; `Button` — тоже автономный объект (конечно, содержащий ссылку делегирования на `Widget`).

С точки зрения паттернов проектирования мы не используем одно имя метода `render(..)` в двух объектах (как предполагается для классов), а выбираем разные имена (`insert(..)` и `build(..)`), которые лучше описывают то, что делает метод. Методы инициализации называются `init(..)` и `setup(..)` по тем же причинам.

Мало того что паттерн проектирования «делегирование» предполагает использование различающихся и более содержательных имен (вместо общих и более универсальных имен) — с OLOO это позволяет избежать уродства явных псевдополиморфных вызовов (`Widget.call` и `Widget.prototype.render.call`), что наглядно демонстрируют простые, относительные, делегированные вызовы `this.init(..)` и `this.insert(..)`.

На уровне синтаксиса здесь нет конструкторов, прототипов или `new`, поскольку все это по сути относится к категории избыточной синтаксической шелухи.

Читатели, внимательно следившие за происходящим, могли заметить, что там, где раньше был всего один вызов (`var btn1 = new Button(..)`), теперь появились два вызова (`var btn1 = Object.create(Button)` и `btn1.setup(..)`). На первый взгляд это кажется недостатком (кода стало больше).

Тем не менее даже это является признаком профессионального кода в стиле OLOO по сравнению с классическим кодом в стиле прототипов. Как?

С конструкторами классов вы вынуждены (ну или как минимум настоятельно рекомендуется) выполнять конструирование и инициализацию за один шаг. Тем не менее во многих ситуациях возможность раздельного выполнения этих двух шагов (как это делается с OLOO) обладает большей гибкостью.

Допустим, вы создаете все свои экземпляры в пуле в начале выполнения программы, но откладываете их инициализацию до того

момента, когда экземпляры извлекаются из пула для использования. Мы разместили два вызова рядом друг с другом в коде, но конечно, они могут происходить в разное время и в разных частях кода.

OLOO лучше поддерживает принцип разделения обязанностей, согласно которому создание и инициализация не обязательно объединены в одну операцию.

Упрощение архитектуры

Кроме того что OLOO способствует созданию ощутимо более простого (и более гибкого) кода, делегирование поведения как паттерн может упростить архитектуру кода. Рассмотрим еще один пример, демонстрирующий упрощение общей архитектуры при использовании OLOO.

В рассматриваемой ситуации существуют два объекта-контроллера: один предназначен для формы входа на веб-странице, а другой обеспечивает аутентификацию (обмен данными) на сервере.

Нам понадобится вспомогательная функция для проведения AJAX-взаимодействий с сервером. В коде будет использоваться jQuery (хотя подойдет и любой другой фреймворк), так как jQuery не только обеспечит использование AJAX, но и вернет ответ в стиле «обещания» (промис, promise), который можно будет прослушивать в вызывающем коде с `.then(..)`.

Промисы здесь не рассматриваются, но эта тема будет изложена в одной из будущих книг серии.

Следуя типичному паттерну проектирования «класс», мы разместим базовую функциональность задачи в классе `Controller`, а затем определим два дочерних класса — `LoginController` и `AuthController`. Оба класса наследуют от `Controller` и специализируют часть его базового поведения:

```
// Родительский класс
function Controller() {
    this.errors = [];
}
Controller.prototype.showDialog = function(title,msg) {
    // вывод заголовка и сообщения в диалоговом окне
};
Controller.prototype.success = function(msg) {
    this.showDialog( "Success", msg );
};
Controller.prototype.failure = function(err) {
    this.errors.push( err );
    this.showDialog( "Error", err );
};

// Дочерний класс
function LoginController() {
    Controller.call( this );
}
// Связывание дочернего класса с родительским
LoginController.prototype =
    Object.create( Controller.prototype );
LoginController.prototype.getUser = function() {
    return document.getElementById( "login_username" ).value;
};
LoginController.prototype.getPassword = function() {
    return document.getElementById( "login_password" ).value;
};
LoginController.prototype.validateEntry = function(user,pw) {
    user = user || this.getUser();
    pw = pw || this.getPassword();

    if (!(user && pw)) {
        return this.failure(
```

```
            "Please enter a username & password!"
        );
    }
    else if (pw.length < 5) {
        return this.failure(
            "Password must be 5+ characters!"
        );
    }

    // управление передано сюда? Проверка прошла успешно!
    return true;
};
// Переопределение для расширения базовой версии `failure()`
LoginController.prototype.failure = function(err) {
    // вызов "super"
    Controller.prototype.failure.call(
       this,
       "Login invalid: " + err
    );
};

// Дочерний класс
function AuthController(login) {
    Controller.call( this );
    // кроме наследования также понадобится композиция
    this.login = login;
}
// Связывание дочернего класса с родительским
AuthController.prototype =
   Object.create( Controller.prototype );
AuthController.prototype.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.prototype.checkAuth = function() {
    var user = this.login.getUser();
    var pw = this.login.getPassword();

    if (this.login.validateEntry( user, pw )) {
```

```
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.success.bind( this ) )
        .fail( this.failure.bind( this ) );
    }
};
// Переопределение для расширения базовой версии `success()`
AuthController.prototype.success = function() {
    // вызов "super"
    Controller.prototype.success.call( this, "Authenticated!" );
};
// Переопределение для расширения базовой версии `failure()`
AuthController.prototype.failure = function(err) {
    // Вызов "super"
    Controller.prototype.failure.call(
       this,
       "Auth Failed: " + err
    );
};

var auth = new AuthController(
    // кроме наследования также понадобится композиция
    new LoginController()
);
auth.checkAuth();
```

Имеется базовое поведение, общее для всех контроллеров: `success(..)`, `failure(..)` и `showDialog(..)`. Дочерние классы `LoginController` и `AuthController` переопределяют `failure(..)` и `success(..)` для расширения поведения базового класса по умолчанию. Также следует заметить, что `AuthController` необходим экземпляр `LoginController` для взаимодействия с формой входа, поэтому такой экземпляр становится свойством данных.

Также следует упомянуть о том, что мы решили применить щепотку композиции поверх наследования. `AuthController` необходима информация о `LoginController`, поэтому мы создаем экземп-

ляр (`new LoginController()`) и создаем свойство `this.login` для обращения к нему, чтобы экземпляр `AuthController` мог активизировать поведение `LoginController`.

Возможно, у вас появится соблазн заставить `AuthController` наследовать от `LoginController` или наоборот, чтобы организовать *виртуальную композицию* по цепочке наследования. В данном случае наследование классов очевидным образом не подходит для предметной области, потому что ни AuthController, ни LoginController не специализируют базовое поведение друг друга, так что наследование между ними выглядит нелогично (если только классы не являются единственно доступным паттерном проектирования). Вместо этого мы применили простую *композицию*, и теперь объекты могут взаимодействовать, при этом пользуясь наследованием от базового родителя `Controller`.

Если вы знакомы с объектно-ориентированным проектированием, такая архитектура должна выглядеть вполне знакомой и естественной.

Расставание с классами

Но так ли уж необходимо моделировать задачу с родительским классом `Controller`, двумя дочерними классами и композицией? Нельзя ли воспользоваться делегированием поведения в стиле OLOO и получить *намного более простую* архитектуру? Да!

```
var LoginController = {
    errors: [],
    getUser: function() {
        return document.getElementById(
           "login_username"
        ).value;
    },
```

```
    getPassword: function() {
        return document.getElementById(
           "login_password"
        ).value;
    },
    validateEntry: function(user,pw) {
        user = user || this.getUser();
        pw = pw || this.getPassword();

        if (!(user && pw)) {
            return this.failure(
               "Please enter a username & password!"
            );
        }
        else if (pw.length < 5) {
            return this.failure(
               "Password must be 5+ characters!"
            );
        }

        // управление передано сюда? Проверка прошла успешно!
        return true;
    },
    showDialog: function(title,msg) {
        // вывести сообщение для пользователя в диалоговом окне
    },
    failure: function(err) {
        this.errors.push( err );
        this.showDialog( "Error", "Login invalid: " + err );
    }
};

// Связывание `AuthController` для делегирования
`LoginController`
var AuthController = Object.create( LoginController );

AuthController.errors = [];
AuthController.checkAuth = function() {
    var user = this.getUser();
    var pw = this.getPassword();
    if (this.validateEntry( user, pw )) {
```

```
        this.server( "/check-auth",{
            user: user,
            pw: pw
        } )
        .then( this.accepted.bind( this ) )
        .fail( this.rejected.bind( this ) );
    }
};
AuthController.server = function(url,data) {
    return $.ajax( {
        url: url,
        data: data
    } );
};
AuthController.accepted = function() {
    this.showDialog( "Success", "Authenticated!" )
};
AuthController.rejected = function(err) {
    this.failure( "Auth Failed: " + err );
};
```

Так как `AuthController` всего лишь объект (как и `LoginController`), не нужно создавать экземпляр (типа `new AuthController()`) для выполнения его задачи. Необходимо совсем немного:

```
AuthController.checkAuth();
```

Конечно, когда вам понадобится создать один или несколько дополнительных объектов в цепочке делегирования при использовании OLOO, это делается достаточно просто и тоже не требует ничего похожего на создание экземпляров классов:

```
var controller1 = Object.create( AuthController );
var controller2 = Object.create( AuthController );
```

С делегированием поведения `AuthController` и `LoginController` представляют собой обычные объекты с одинаковыми правами; они не связаны иерархическими связями типа «родитель/потомок» в программировании, ориентированном на использование

классов. Мы достаточно произвольно выбрали делегирование от `AuthController` к `LoginController`; делегирование с таким же успехом работало бы в обратном направлении.

Главная особенность второго листинга заключается в том, что в архитектуре используются только две сущности (`LoginController` и `AuthController`), а не три, как прежде.

Нам не нужен базовый класс `Controller` с «общим» поведением этих двух сущностей, потому что механизм делегирования обладает достаточной мощью, для того чтобы предоставить всю необходимую функциональность. Как было отмечено ранее, не нужно создавать экземпляры классов для работы с ними, потому что никаких классов нет, есть только сами объекты. Более того, нет необходимости в композиции, так как делегирование позволяет двум объектам взаимодействовать, дополняя функциональность друг друга, по мере необходимости.

Наконец, мы избежали полиморфных ловушек проектирования, ориентированного на использование классов, и отказались от одинаковых имен `success(..)` и `failure(..)` в обоих объектах, поскольку это потребовало бы уродливых конструкций явного псевдополиморфизма. Вместо этого в `AuthController` были выбраны имена `accepted()` и `rejected(..)`, чуть более подходящие для их конкретных задач.

Резюме: мы получаем ту же функциональность, но существенно более простой архитектурой. Такова сила программирования в стиле OLOO и паттерна проектирования «*делегирование поведения*».

Более приятный синтаксис

Одной из причин, по которым конструкция `class` в ES6 выглядит обманчиво привлекательно (о том, почему ее стоит избегать, рас-

сказано в приложении Г), является сокращенный синтаксис объявления методов классов:

```
class Foo {
    methodName() { /* .. */ }
}
```

Из объявления пропадает слово `function`, все разработчики JS в восторге!

Возможно, вы заметили, что в представленном ранее синтаксисе OLOO неоднократно встречается слово `function`. И наверное, вас это огорчило, ведь лишние слова отходят от цели упрощения кода OLOO. Тем не менее это вовсе не обязательно.

ES6 позволяет использовать компактные объявления методов в любых объектных литералах, так что объект в стиле OLOO может быть объявлен следующим образом (такое же синтаксическое удобство, как основной синтаксис `class`):

```
var LoginController = {
    errors: [],
    getUser() { // Смотрите, здесь нет `function`!
        // ...
    },
    getPassword() {
        // ...
    }
    // ...
};
```

Пожалуй, единственное различие заключается в том, что объектные литералы все еще требуют разделителей-запятых между элементами, а в синтаксисе `class` они не нужны. В общей картине это второстепенная подробность.

Более того, в ES6 более громоздкий синтаксис (вроде определения `AuthController`), при котором свойства назначаются по отдель-

ности без использования объектных литералов, может быть переписан с объектными литералами (для использования компактных методов), а свойство `[[Prototype]]` этого объекта может быть изменено вызовом `Object.setPrototypeOf(..)`:

```
// удобный синтаксис объектных литералов с компактными методами!
var AuthController = {
    errors: [],
    checkAuth() {
        // ...
    },
    server(url,data) {
        // ...
    }
    // ...
};

// ТЕПЕРЬ `AuthController` связывается для делегирования
`LoginController`
Object.setPrototypeOf( AuthController, LoginController );
```

Стиль OLOO в синтаксисе ES6 с компактными методами получается намного более удобным, чем прежде (хотя даже тогда он был намного проще и приятнее, чем классический код с прототипом). Не обязательно выбирать классы (сложность), чтобы использовать удобный объектный синтаксис!

Нелексичность

У компактных методов есть один недостаток, нетривиальный, но важный. Рассмотрим следующий код:

```
var Foo = {
    bar() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

Следующее синтаксическое разложение показывает, как будет работать код:

```
var Foo = {
    bar: function() { /*..*/ },
    baz: function baz() { /*..*/ }
};
```

Видите отличие? Сокращенная запись `bar()` становится *анонимным функциональным выражением* (`function()..`), присоединенным к свойству `bar`, потому что сам объект функции не имеет идентификатора. Сравните с *именованным функциональным выражением*, заданным вручную (`function baz()..`), которое имеет лексический идентификатор `baz` помимо присоединения к свойству `.baz`.

Что из того? Три главных недостатка *анонимных функциональных выражений* подробно рассмотрены в первой части этой книги. Кратко повторю их для сравнения с компактной сокращенной записью методов.

Отсутствие идентификатора у анонимной функции:

1. Усложняет отладку трассировки стека.
2. Усложняет автоссылки (рекурсия, связывание событий и т. д.).
3. Затрудняет (в некоторой степени) понимание кода.

Пункты 1 и 3 не относятся к компактным методам.

И хотя в разложении используется анонимное функциональное выражение, для которого обычно в трассировку стека имя не включается, компактные методы присваивают внутреннему свойству `name` объекта функции соответствующее значение, так что оно может использоваться трассировкой стека (хотя это поведение зависит от реализации, так что оно не гарантировано).

К сожалению, пункт 2 так и остается недостатком компактных методов. У них нет лексического идентификатора, который мог бы использоваться в качестве автоссылки. Пример:

```
var Foo = {
    bar: function(x) {
        if (x < 10) {
            return Foo.bar( x * 2 );
        }
        return x;
    },
    baz: function baz(x) {
        if (x < 10) {
            return baz( x * 2 );
        }
        return x;
    }
};
```

В данном примере «ручной» ссылки `Foo.bar(x*2)` в общем-то достаточно, но найдется немало ситуаций, в которых этот вариант невозможен, например, если функция совместно используется при делегировании между разными объектами, при использовании связывания `this` и т. д. Вы бы предпочли использовать настоящую автоссылку, и самым удобным кандидатом стал бы идентификатор объекта функции.

Просто помните об этом недостатке компактных методов, и если вы столкнетесь с подобными проблемами из-за отсутствия автоссылок, просто откажитесь от компактного синтаксиса *только для этого объявления* в пользу формы ручного объявления *именованных функциональных выражений*: `baz: function baz(){..}`.

Интроспекция

Если вы провели достаточно времени за программированием, ориентированным на использование классов (с JS или с другими

языками), вероятно, вам встречался термин «*интроспекция*»: анализ экземпляра с целью определения того, каким объектом он является. Интроспекция используется с экземплярами классов прежде всего для получения информации о структуре/возможностях объекта на основании *того, как он был создан*.

В следующем фрагменте `instanceof` (см. главу 10) используется для интроспекции объекта `a1` и получения информации о его возможностях:

```
function Foo() {
    // ...
}
Foo.prototype.something = function(){
    // ...
}
var a1 = new Foo();
// позднее
if (a1 instanceof Foo) {
    a1.something();
}
```

Поскольку `Foo.prototype` (не `Foo`) входит в цепочку `[[Prototype]]` (см. главу 10) объекта `a1`, оператор `instanceof` (как ни странно) решает сообщить нам, что `a1` является экземпляром «класса» `Foo`. При наличии такой информации можно считать, что `a1` обладает функциональностью, описанной классом `Foo`.

Конечно, никакого класса `Foo` нет, есть самая обычная функция `Foo`, содержащая ссылку на произвольный объект (`Foo.prototype`), с которым `a1` связывается через делегирование. По своему синтаксису оператор `instanceof` делает вид, что он анализирует отношения между `a1` и `Foo`, но на самом деле он просто сообщает, связаны ли `a1` и (произвольный объект, на который указывает ссылка) `Foo.prototype`.

Семантическая путаница (и косвенный характер) синтаксиса `instanceof` означает, что при использовании интроспекции на базе

instanceof для получения информации о том, связан ли объект a1 с указанным объектом, нужно иметь функцию, которая содержит ссылку на этот объект; вы не можете просто спросить, связаны ли между собой два объекта.

Вспомните абстрактный пример Foo/Bar/b1, приведенный ранее в этой главе; приведем его в сокращенном виде:

```
function Foo() { /* .. */ }
Foo.prototype...

function Bar() { /* .. */ }
Bar.prototype = Object.create( Foo.prototype );

var b1 = new Bar( "b1" );
```

Для выполнения интроспекции в этом примере в семантике instanceof и .prototype можно воспользоваться следующими проверками:

```
// связывание `Foo` и `Bar` друг с другом
Bar.prototype instanceof Foo; // true
Object.getPrototypeOf( Bar.prototype )
   === Foo.prototype; // true
Foo.prototype.isPrototypeOf( Bar.prototype ); // true

// связывание `b1` с `Foo` и `Bar`
b1 instanceof Foo; // true
b1 instanceof Bar; // true
Object.getPrototypeOf( b1 ) === Bar.prototype; // true
Foo.prototype.isPrototypeOf( b1 ); // true
Bar.prototype.isPrototypeOf( b1 ); // true
```

Честно признаю, что все это не идеально. Например, (с классами) вам бы интуитивно хотелось использовать конструкции вида Bar instanceof Foo (потому что легко запутаться в термине «экземпляр» и думать, что он подразумевает «наследование»), но в JS такие сравнения бессмысленны. Вместо этого приходится использовать запись Bar.prototype instanceof Foo.

Другой распространенный, но, возможно, менее надежный паттерн интроспекции, которому многие разработчики отдают предпочтение перед `instanceof`, — так называемая «утиная типизация». Термин происходит от поговорки «Если что-то выглядит как утка и крякает как утка, то, скорее всего, это и есть утка».

Пример:

```
if (a1.something) {
    a1.something();
}
```

Вместо того чтобы анализировать отношения между `a1` и объектом, содержащим делегируемую функцию `something()`, мы предполагаем, что проверка условия `a1.something` означает, что `a1` способен вызвать метод `.something()` (независимо от того, находится ли он непосредственно в `a1` или делегируется другому объекту). Само по себе это предположение не слишком рискованно.

Но «утиная типизация» часто расширяется для других предположений о возможностях объекта, кроме проверяемых. Конечно, такие предположения вносят в проверку дополнительный риск (то есть снижают надежность архитектуры).

Один заметный пример «утиной типизации» встречается в промисах `Promise` ES6 (которые, как упоминалось в предшествующей врезке, в этой книге не рассматриваются).

По различным причинам может потребоваться определить, представляет ли ссылка на произвольный объект `Promise`, но для этого проверяется, содержит ли объект функцию `then()`. Другими словами, если объект содержит метод `then()`, то промисы ES6 безусловно предполагают, что этот объект «`then`-совместим», а следовательно, его поведение соответствует стандартному поведению `Promise`.

Если у вас имеется посторонний объект (не `Promise`), который по какой-то причине содержит метод `then()`, старайтесь держаться подальше от механизма ES6 `Promise`, чтобы избежать ошибочных предположений. Этот пример наглядно демонстрирует риск «утиной типизации». Не увлекайтесь такими решениями и применяйте их только в управляемых условиях.

Возвращаясь к коду в стиле OLOO, код интроспекции получается куда более «чистым». Вспомните OLOO-пример `Foo/Bar/b1`, приведенный ранее в этой главе (приводится в сокращении):

```
var Foo = { /* .. */ };

var Bar = Object.create( Foo );
Bar...

var b1 = Object.create( Bar );
```

При использовании подхода OLOO, при котором есть только обычные объекты, связанные посредством делегирования `[[Prototype]]`, можно воспользоваться относительно простой интроспекцией:

```
// связывание `Foo` и `Bar` друг с другом
Foo.isPrototypeOf( Bar ); // true
Object.getPrototypeOf( Bar ) === Foo; // true

// связывание `b1` с `Foo` и `Bar`
Foo.isPrototypeOf( b1 ); // true
Bar.isPrototypeOf( b1 ); // true
Object.getPrototypeOf( b1 ) === Bar; // true
```

Здесь `instanceof` не используется, потому что попытки представить, что это имеет какое-то отношение к классам, только создают путаницу. Теперь мы просто задаем (неформально изложенный) вопрос: «Ты являешься моим прототипом?» Отпадает необходимость в косвенных конструкциях вида `Foo.prototype` или до отвращения громоздких `Foo.prototype.isPrototypeOf(..)`.

Я думаю, можно честно признать, что эти проверки значительно менее сложны и запутанны, чем предыдущий набор проверок интроспекции. И снова мы видим, что в JavaScript код в стиле OLOO проще кода в стиле классов (но сохраняет всю его мощь).

Итоги

Классы и наследование — паттерн проектирования, который вы можете выбрать или не выбрать в архитектуре своего программного продукта. Многие разработчики беспрекословно считают, что классы являются единственным (правильным) способом организации кода, но в этой главе было показано, что существует другой, очень мощный, но реже обсуждаемый паттерн: делегирование поведения.

Делегирование поведения предполагает, что объекты находятся на одном уровне иерархии и делегируют поведение друг другу (в отличие от отношений «родитель/потомок», типичных для классов). Механизм `[[Prototype]]` в JavaScript по самой своей природе является механизмом делегирования поведения. Это означает, что вы можете либо прикладывать усилия, чтобы реализовать механику классов поверх JS (см. главы 9 и 10), либо просто принять естественное состояние `[[Prototype]]` как механизм делегирования.

Проектирование кода только с одними объектами не только упрощает синтаксис, но и приводит к более простой архитектуре кода.

OLOO (Objects Linked to Other Objects) — стиль программирования, при котором объекты создаются и связываются напрямую без участия абстракции классов. OLOO вполне естественно реализует делегирование поведения на базе `[[Prototype]]`.

Приложение Г. Классы ES6

Если из глав 9–11 и можно сделать какой-то краткий вывод, то это вывод о том, что классы являются необязательным паттерном проектирования для кода (а не чем-то данным свыше); более того, часто они довольно неудобно реализуются на таких языках с поддержкой `[[Prototype]]`, как JavaScript.

Неудобство проявляется не только в синтаксисе, хотя он и является значительной его частью. В главах 10 и 11 синтаксическое уродство анализируется достаточно подробно — от длинных ссылок `.prototype`, загромождающих код, до явного псевдополиморфизма (см. главу 9), когда разработчик присваивает методам на разных уровнях цепочки одинаковые имена, а затем пытается реализовать полиморфные обращения из низкоуровневого метода в высокоуровневый метод. Ссылка `.constructor` ошибочно интерпретируется как «тот, кто сконструировал», причем ненадежность для этого определения совмещается с очередным синтаксическим уродством.

Однако проблемы с архитектурой классов лежат намного глубже. В главе 9 указано, что классы в традиционных классово-ориентированных языках фактически выполняют действие копирования из родителя в потомка, а из потомка в экземпляр, тогда как в `[[Prototype]]` действием становится не копирование, а создание ссылки делегирования.

При сравнении с простотой кода в стиле OLOO и делегирования поведения (см. главу 11), которые принимают механизм `[[Prototype]]`, вместо того чтобы отгораживаться от него, классы выделяются в JS как бельмо на глазу.

class

Впрочем, не стоит снова приводить те же доводы. Я остановлюсь на этих вопросах только для того, чтобы напомнить вам об этом сейчас, когда мы обращаемся к механизму классов ES6. Я продемонстрирую, как он работает, а также разберемся, делает ли синтаксис `class` что-нибудь существенное для решения проблем «классов».

Вернемся к примеру `Widget/Button` из главы 6:

```
class Widget {
    constructor(width,height) {
        this.width = width || 50;
        this.height = height || 50;
        this.$elem = null;
    }
    render($where){
        if (this.$elem) {
            this.$elem.css( {
                width: this.width + "px",
                height: this.height + "px"
            } ).appendTo( $where );
        }
    }
}

class Button extends Widget {
    constructor(width,height,label) {
        super( width, height );
        this.label = label || "Default";
```

```
        this.$elem = $( "<button>" ).text( this.label );
    }
    render($where) {
        super( $where );
        this.$elem.click( this.onClick.bind( this ) );
    }
    onClick(evt) {
        console.log( "Button '" + this.label + "' clicked!" );
    }
}
```

Если не считать того, что синтаксис выглядит *приятнее*, какие проблемы решает ES6?

1. Исчезли ссылки на `.prototype`, загромождающие код (*в определенном смысле* — см. ниже).

2. Класс `Button` напрямую объявляется как «наследующий» (то есть расширяющий) от `Widget`, вместо того чтобы использовать `Object.create()` для замены связанного объекта `.prototype` или выполнять присваивание `.__proto__` или `Object.setPrototypeOf(..)`.

3. `super(..)` теперь предоставляет очень полезную функциональность относительного полиморфизма, так что любой метод на одном уровне цепочки может относительно обращаться к одноименному методу на более высоком уровне в цепочке. В частности, таким образом решается проблема, упомянутая во врезке главы 9, о странности конструкторов, не принадлежащих своему классу. В конструкторах `super()` работает именно так, как вы ожидаете.

4. В синтаксисе литералов классов не предусмотрена возможность задания свойств (только методов). Кому-то это может показаться ограничением, но предполагается, что в подавляющем большинстве случаев существование свойств (состояния) в любом месте, кроме завершающих цепочку «экземпляров»,

обычно является ошибкой (так как это состояние неявно становится «общим» для всех «экземпляров»). Итак, можно сказать, что синтаксис `class` защищает вас от ошибок.

5. `extends` позволяет расширять даже встроенные объектные (под) типы, такие как `Array` или `RegExp`, очень естественным образом. Решение этой задачи без `class..extends` издавна считалось делом исключительно сложным и неприятным, таким, с которым способен справиться только самый компетентный автор фреймворков. Теперь эта задача решается вполне тривиально!

Если говорить по справедливости, новый синтаксис предоставляет немало основательных решений многих очевидных (синтаксических) проблем и сюрпризов, с которыми сталкивались разработчики с классическим кодом в стиле прототипов.

Проблемы class

Однако не все так радужно. У «классов» как паттерна проектирования в JS существуют глубокие, принципиально трудноразрешимые проблемы. Во-первых, синтаксис `class` может убедить вас в том, что начиная с ES6 в JS появился новый механизм «классов». Это не так — ключевое слово `class` остается всего лишь синтаксическим удобством на базе существующего механизма `[[Prototype]]` (делегирования).

Это означает, что `class` не выполняет статического копирования определений в момент объявления, как это делается в традиционных языках, ориентированных на использование классов. Если вы измените/замените метод (намеренно или случайно) родительского «класса», это отразится на дочернем «классе» и/или экземплярах: они не получат копии в момент объявления, а все еще будут использовать модель делегирования на базе `[[Prototype]]`:

```
class C {
    constructor() {
        this.num = Math.random();
    }
    rand() {
        console.log( "Random: " + this.num );
    }
}

var c1 = new C();
c1.rand(); // "Random: 0.4324299..."

C.prototype.rand = function() {
    console.log( "Random: " + Math.round( this.num * 1000 ));
};

var c2 = new C();
c2.rand(); // "Random: 867"

c1.rand(); // "Random: 432" -- ой!!!
```

Такое поведение кажется разумным, только если *вы уже знаете* о том, что все строится на основе делегирования, и не ожидаете копирования от «настоящих классов». Таким образом, вы должны задать себе вопрос: почему вы выбираете синтаксис классов для чего-то, столь фундаментально отличающегося от классов?

Разве синтаксис классов ES6 не мешает увидеть и понять различия между традиционными классами и делегированными объектами?

Синтаксис `class` не дает возможности объявлять свойства в классах (только методы). Таким образом, если вам понадобится отслеживать общее состояние для разных экземпляров, в итоге придется прибегнуть к уродливому синтаксису `.prototype`:

```
class C {
    constructor() {
        // необходимо изменить общее состояние,
        // а не задать замещенное свойство
```

```
        // в экземплярах!
        C.prototype.count++;

        // здесь `this.count` работает так, как и ожидалось,
        // благодаря делегированию!
        console.log( "Hello: " + this.count );
    }
}

// добавить свойство для общего состояния
// прямо в объект prototype
C.prototype.count = 0;

var c1 = new C();
// Hello: 1

var c2 = new C();
// Hello: 2

c1.count === 2; // true
c1.count === c2.count; // true
```

Самая большая проблема заключается в том, что этот код изменяет синтаксису классов, раскрывая `.prototype` как подробность реализации.

Но здесь также кроется одна неожиданная ловушка: запись `this.count++` неявно создаст отдельное замещенное свойство `.count` в объектах `c1` и `c2`, вместо того чтобы обновить общее состояние. `class` ничего не делает для того, чтобы решить эту проблему — разве что намекает (отсутствием синтаксической поддержки), что так поступать вообще не следует.

Более того, риск случайного замещения никуда не исчез:

```
class C {
    constructor(id) {
        // ловушка - метод `id()` замещается
        // значением свойства экземпляра
```

```
        this.id = id;
    }
    id() {
        console.log( "Id: " + id );
    }
}

var c1 = new C( "c1" );
c1.id(); // TypeError -- `c1.id` теперь является строкой "c1"
```

Также существуют очень нетривиальные нюансы относительно того, как работает `super`. Можно предположить, что связывание `super` происходит по аналогии со связыванием `this` (см. главу 7), то есть что `super` всегда будет связываться на один уровень выше позиции текущего метода в цепочке `[[Prototype]]`.

Однако по соображениям быстродействия (связывание `this` и так обходится достаточно дорого) `super` не связывается динамически. Связывание происходит отчасти «статически» в момент объявления. Ничего страшного, правда?

Ммм... Возможно. А может, и нет. Если вы, как и большинство разработчиков JS, начинаете с присваивания функций разным объектам (что происходит от определений `class`) разными способами, скорее всего, вы даже не очень понимаете, что во всех этих случаях механизм `super` каждый раз связывается заново.

И в зависимости от того, какие синтаксические решения вы будете использовать при этих присваиваниях, вполне может оказаться, что в каких-то случаях `super` будет связываться неправильно (по крайней мере не так, как вы ожидаете). Может оказаться (на момент написания книги в TC39 еще шло обсуждение этой темы), что вам придется вручную связывать `super` вызовом `toMethod(..)` (по аналогии с тем, как вы вызываете `bind(..)` для `this` — см. главу 7).

Вы привыкли к тому, что можете присваивать методы разным объектам, чтобы *автоматически* использовать динамизм `this` по

правилу *неявного связывания* (см. главу 7). Однако это вряд ли будет относиться к методам, использующим `super`.

Подумайте, что `super` сделает в этом случае (для `D` и `E`):

```
class P {
    foo() { console.log( "P.foo" ); }
}

class C extends P {
    foo() {
        super();
    }
}

var c1 = new C();
c1.foo(); // "P.foo"

var D = {
    foo: function() { console.log( "D.foo" ); }
};

var E = {
    foo: C.prototype.foo
};

// Связывание E с D для делегирования
Object.setPrototypeOf( E, D );

E.foo(); // "P.foo"
```

Если вы думаете (вполне разумно), что `super` будет динамически связываться в момент вызова, можно ожидать, что `super()` будет автоматически определять, что `E` делегирует `D`, так что для `E.foo()` с использованием `super()` будет вызываться `D.foo()`.

Это не так. По соображениям практического быстродействия для `super` применяется не *позднее* (динамическое) связывание, как для `this`. Вместо этого оно будет определяться в момент вызова по

`[[HomeObject]].[[Prototype]]`, где `[[HomeObject]]` статически связывается в момент создания.

В этом конкретном случае `super()` все равно разрешается в `P.foo()`, поскольку `[[HomeObject]]` для метода остается `C`, а `C.[[Prototype]]` является `P`.

Возможно, такие проблемы можно решить вручную. Использование `toMethod(..)` для связывания/повторного связывания `[[HomeObject]]` (наряду с назначением `[[Prototype]]` этого объекта) вроде бы работает в этом сценарии:

```
var D = {
    foo: function() { console.log( "D.foo" ); }
};

// Связывание E с D для делегирования
var E = Object.create( D );

// Вручную связать `[[HomeObject]]` для foo
// с `E`, а `E.[[Prototype]]` с `D`, поэтому
// `super()` будет `D.foo()`
E.foo = C.prototype.foo.toMethod( E, "foo" );

E.foo(); // "D.foo"
```

`toMethod(..)` клонирует метод и получает `homeObject` в первом параметре (поэтому мы передаем `E`), а во втором параметре (не обязательном) задается имя нового метода (которому оставляется имя `"foo"`).

Остается увидеть, существуют ли другие проблемы граничных случаев, с которыми может столкнуться разработчик помимо этого сценария. В любом случае вам придется действовать очень осторожно и понимать, в каких местах движок автоматически

определяет `super` за вас и в каких местах вы должны позаботиться о нем вручную. Кому это понравится?

Статический > динамический?

Но самая большая проблема конструкции `class` в ES6 заключается в том, что разнообразные ловушки в каком-то смысле навязывают вам синтаксис, который вроде бы подразумевает (по аналогии с традиционными классами), что после объявления класса он становится статическим определением некой сущности (экземпляра, который будет создан в будущем). Из виду полностью теряется тот факт, что `C` является объектом, конкретной сущностью, с которой можно взаимодействовать напрямую.

В традиционных языках, ориентированных на использование классов, вам не удастся изменить определение класса в будущем, так что паттерн проектирования «класс» не предполагает такой возможности. Но одна из самых выдающихся особенностей языка JS — его динамизм и тот факт, что определение любого объекта (если только он не сделан неизменяемым) является подвижным и изменяемым.

`class` вроде бы подразумевает, что так поступать не следует, заставляя вас использовать более уродливый синтаксис `.prototype`, помнить обо всех ловушках с `super` и т. д. `class` также предоставляет минимальную поддержку для решения проблем, которые может создать такой динамизм.

Иначе говоря, `class` словно говорит вам: «Динамизм — это слишком сложно. Наверное, тебе это не нужно. Вот тебе синтаксис, который кажется статическим; программируй соответствующим образом».

По этим причинам синтаксис `class` ES6 притворяется хорошим решением синтаксических проблем, но в действительности он лишь наводит туман в ущерб JS и четкому, ясному пониманию.

Если вы используете функцию `.bind(..)` для создания жестко связанной функции (см. главу 7), то созданная функция не может создавать подклассы ключевым словом ES6 `extend`, как обычные функции.

Итоги

Ключевое слово `class` неплохо имитирует решение проблем паттерна «класс/наследование» в JS. Но на самом деле `class` делает обратное: он маскирует многие проблемы и вводит другие неочевидные, но опасные ловушки.

Синтаксис `class` вносит свой вклад в непреходящую путаницу с «классами» в JavaScript, которая преследует язык почти два десятилетия. В некоторых отношениях он больше задает вопросов, чем отвечает на них, и выглядит противоестественной заплаткой поверх элегантной простоты механизма `[[Prototype]]`.

Резюме: если синтаксис ES6 `class` усложняет нормальную работу с `[[Prototype]]` и скрывает важнейший аспект объектного механизма JS — «живые» ссылки делегирования между объектами, не следует ли из этого, что синтаксис `class` создает больше проблем, чем решает их, и его стоит просто рассматривать как антипаттерн?

Я не смогу ответить на этот вопрос за вас. Но надеюсь, что эта книга рассматривает вопрос на более глубоком уровне, чем вы когда-либо изучали его ранее, и дает вам информацию, необходимую для того, чтобы вы *ответили на него самостоятельно*.

Об авторе

Кайл Симпсон — евангелист Open Web из Остина (штат Техас), большой энтузиаст всего, что касается JavaScript. Автор нескольких книг, преподаватель, спикер и участник/лидер проектов с открытым кодом.

К. Симпсон

{Вы не знаете JS}
Замыкания и объекты

Перевел с английского Е. Матвеев

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Бульченко*
Художественный редактор	*В. Мостипан*
Корректоры	*С. Беляева, М. Молчанова*
Верстка	*Л. Егорова*

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: 05.2019. 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 14.05.19. Формат 60x90/16. Бумага офсетная. Усл. п. л. 21,000.
Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР»
WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanovaa@piter.com
Подробная информация здесь: http://www.piter.com/page/avtoru

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок: *тел./факс: (812) 703-73-73; e-mail: sales@piter.com*

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: *на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216; e-mail: books@piter.com*

Вопросы по продаже электронных книг: *тел.: (812) 703-73-74, доб. 6217; e-mail: kuznetsov@piter.com*