O'REILLY®

3-е издание
Рассмотрен стандарт ES6!

Изучаем JavaScript

РУКОВОДСТВО ПО СОЗДАНИЮ СОВРЕМЕННЫХ ВЕБ-САЙТОВ

Этан Браун

O'REILLY®

Изучаем JavaScript

Сейчас самое время изучить *JavaScript*. После выхода последней спецификации *JavaScript — ECMAScript 6.0 (ES6)* — научиться создавать высококачественные приложения на этом языке стало проще, чем когда-либо ранее. Эта книга знакомит программистов (любителей и профессионалов) со спецификацией ES6 наряду с некоторыми связанными с ней инструментальными средствами и методиками на сугубо практической основе.

Этан Браун, автор книги *Web Development with Node and Express*, излагает не только простые и понятные темы (переменные, ветвление потока, массивы), но и более сложные концепции, такие как функциональное и асинхронное программирование. Вы узнаете, как создавать мощные и эффективные веб-приложения для работы на клиенте или сервере *Node.js*.

- Используйте ES6 для транскомпиляции в переносимый код ES5
- Преобразуйте данные в формат, который может использовать *JavaScript*
- Усвойте основы и механику применения функций *JavaScript*
- Изучите объекты и объектно-ориентированное программирование
- Ознакомьтесь с новыми концепциями, такими как итераторы, генераторы и прокси-объекты
- Преодолейте сложности асинхронного программирования
- Используйте объектную модель документа для приложений, выполняемых в браузере
- Изучите основы применения платформы *Node.js* для разработки серверных приложений

"Всем разработчикам действительно пришло время изучить JS. Под изучением я не имею в виду примитивное «Я получил некий работоспособный код». Эта книга — куда глубже и обеспечивает именно то изучение, в котором все мы нуждаемся!"

Кайл Симпсон *(Kyle Simpson)*, автор серии *You Don't Know JS*

"Хорошо написанное сжатое введение в *JavaScript*, включая *ECMAScript 6*".

Аксель Роушмайер *(Axel Rauschmayer)*, автор *Speaking JavaScript*

Этан Браун — директор интерактивного маркетингового агентства Engineering at Pop Art, в котором он отвечает за архитектуру и реализацию веб-сайтов и веб-служб для любых клиентов, от малых предприятий до транснациональных компаний. Этан имеет более чем 20-летний стаж программирования.

ПРОГРАММИРОВАНИЕ ДЛЯ ВЕБ / JAVASCRIPT

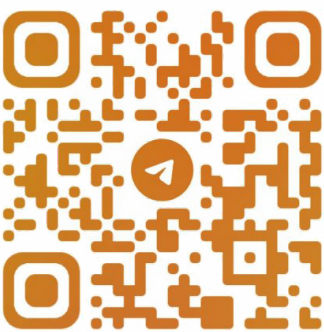
ДИАЛЕКТИКА
www.dialektika.com

Twitter: @oreillymedia
facebook.com/oreilly

ISBN 978-5-9908463-9-5

9 785990 846395 17002

Изучаем JavaScript



Learning JavaScript

THIRD EDITION

Ethan Brown

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

Изучаем JavaScript

РУКОВОДСТВО ПО СОЗДАНИЮ СОВРЕМЕННЫХ ВЕБ-САЙТОВ

3-Е ИЗДАНИЕ

Этан Браун

Диалектика

Москва • Санкт-Петербург • Киев
2017

ББК 32.973.26-018.2.75
Б87
УДК 681.3.07

Компьютерное издательство "Диалектика"

Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция *В.А. Коваленко*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Браун, Этан.

Б87 Изучаем JavaScript: руководство по созданию современных веб-сайтов, 3-е изд. : Пер. с англ. — СпБ. : ООО "Альфа-книга", 2017. — 368 с. : ил. — Парал. тит.

ISBN 978-5-9908463-9-5 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Этан Браун

Изучаем JavaScript
Руководство по созданию современных веб-сайтов
3-е издание

Литературный редактор *Л.Н. Красножон*
Верстка *О.В. Мишутина*
Художественный редактор *Е.П. Дынник*
Корректор *Л.А. Гордиенко*

Подписано в печать 23.03.2017. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 23,0. Уч.-изд. л. 17,8.
Тираж 500 экз. Заказ № 2261

Отпечатано в АО «Первая Образцовая типография»
Филиал «Чеховский Печатный Двор»
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1

ООО "Альфа-книга", 195027, Санкт-Петербург, Магнитогорская ул., д. 30

ISBN 978-5-9908463-9-5 (рус.)

ISBN 978-1-491-91491-5 (англ.)

Оглавление

Содержание

Для Марка — истинного друга и собрата.

Об авторе

Этан Браун — директор интерактивного маркетингового агентства Engineering at Pop Art, в котором он отвечает за архитектуру, реализацию веб-сайтов и веб-служб для любых клиентов, от малых предприятий до транснациональных компаний. Этан обладает более чем 20-летним стажем программирования, начиная со встраивания в веб и заканчивая семейством JavaScript как веб-платформой будущего.

Изображение на обложке

Животное на обложке книги — это детеныш черного носорога (*Diceros bicornis*). Черный носорог — это один из двух видов африканских носорогов. Весящий до половины тонны, он меньше своего собрата — белого носорога. Черные носороги живут в саванне, на открытой лесистой местности и в горных лесах нескольких небольших областей Южной, Юго-Западной, Центральной и Восточной Африки. Они предпочитают жить поодиночке и агрессивно защищают свою территорию.

Внешним отличием черного носорога от белого является форма верхней губы: у черного носорога она заострена и свисает хоботком над нижней. С помощью этой губы животное захватывает листву с веток кустарника. Это позволяет ему есть более грубую растительность, чем другие травоядные животные.

Черные носороги — непарнокопытные животные, т.е. у них по три пальца на каждой ноге. У них толстая серая кожа без шерсти. Одной из главных отличительных особенностей носорога являются два рога, фактически состоящих из слипшихся волос, а не из кости. Носорог использует их для защиты от львов, тигров и гиен, а также для привлечения особей противоположного пола. Ритуал ухаживания зачастую довольно груб и рога могут нанести серьезные раны.

Впоследствии самцы и самки носорогов никаких контактов не поддерживают. Период беременности составляет 14–18 месяцев, а молоком детеныши кормятся в течение года, хотя они в состоянии есть растительную пищу почти сразу же после рождения. Связь между матерью и ее теленком может продолжаться четыре года, пока он оставит ее.

Охота на носорогов поставила их на грань исчезновения. По оценкам ученых сто лет назад популяция черных носорогов в Африке составляла порядка миллиона особей, а сейчас это число сократилось до 2400. Сегодня в опасности находятся все пять оставшихся видов, включая индийского, яванского и суматранского носорогов. Людей не зря считают самыми свирепыми хищниками.

Большинство животных с обложек O'Reilly находятся в опасности; все они важны для мира. Чтобы узнать больше, как можно им помочь, обратитесь по адресу `animals.oreilly.com`. Изображение на этой обложке взято из книги *Natural History* Джона Кассела (John Cassell).

Хотя это моя вторая книга по технологиям JavaScript, роль эксперта по JavaScript меня все еще несколько смущает. Подобно большинству программистов, я имел некое предубеждение относительно JavaScript вплоть до примерно 2012 года. Хотя моя позиция резко изменилась, я все еще чувствую легкое смущение.

Причина моего предубеждения была обычной: я считал JavaScript "игрушечным" языком (не изучив его толком, а потому и не зная, о чем говорю), опасным, сырым, используемым безграмотными программистами-любителями. В обеих этих причинах была некая доля истины. Спецификация ES6 была разработана очень быстро, и даже ее изобретатель, Брендан Айк (Brendan Eich), признает, что есть вещи, которых он в первое время не понимал, а когда понял, уже слишком много людей полагались на проблематичное для него поведение, чтобы эффективно его изменить (но покажите мне язык, который не страдал бы от подобных проблем). Что касается второй причины, JavaScript внезапно *сделал* программирование доступным. Мало того что браузер есть у всех, так еще и усилий для создания веб-сайтов с использованием JavaScript, которые быстро множились бы в Интернете, необходимо совсем немного. Люди учатся методом проб и ошибок, читая коды друг друга и (в очень многих случаях) подражая плохо написанному коду безо всякого понимания.

Я рад, что узнал о JavaScript достаточно, чтобы понять, что этот (далеко не игрушечный) язык разработан на чрезвычайно прочном фундаменте и отличается мощью, гибкостью и выразительностью. Я также рад, что уловил доступность, обеспечиваемую языком JavaScript. Я, конечно, не испытываю никакой враждебности к любителям: все должны с чего-то начинать, программирование — выгодный навык, и у карьеры программиста есть много преимуществ.

Начинающему программисту, любителю, я могу сказать, что нет ничего позорного в том, чтобы быть любителем. Есть некий позор в том, чтобы *оставаться* любителем (если, конечно, вы сделали программирование своей профессией). Если нужен опыт в программировании, то *приобретайте его*. Изучите все, что сможете, все доступные первоисточники, какие найдете. Не будьте предвзятыми и (возможно, это важнее всего) подвергайте сомнению все. Расспрашивайте каждого эксперта. Расспрашивайте каждого опытного программиста. Постоянно спрашивайте "Почему?"

В этой книге по большей части я пытался придерживаться "фактов" JavaScript, но полностью избежать собственного мнения невозможно. Когда я выражаю

собственное мнение, я так и говорю. Вы вполне можете не соглашаться с ним и придерживаться мнения других опытных разработчиков.

Вы изучаете JavaScript в самый подходящий момент. Веб вышел из младенческого возраста (с технической точки зрения), а веб-разработка, без сомнения, — больше не Дикий Запад, которым она была лет 5–10 назад. Такие стандарты, как HTML5 и ES6, облегчают изучение веб-разработки и упрощают разработку высококачественных приложений. Платформа Node.js делает JavaScript доступным и вне браузера; теперь это вполне подходящий выбор для системных сценариев, разработки приложений рабочего стола, приложений для веб-серверов и даже для встраиваемых приложений. Конечно, я не имел такого удовольствия в программировании, поскольку начал лишь с середины 1980-х годов.

Краткая история JavaScript

Язык JavaScript был разработан Бренданом Айком из корпорации Netscape Communications Corporation в 1995 году. Его первая разработка была весьма скороспелой, и критики JavaScript по большей части порицали недостатки предварительного планирования во время его разработки. Однако Брендан Айк не был дилетантом: у него был серьезный опыт в информатике, и он заложил в JavaScript на удивление сложные и передовые идеи. Так или иначе, он опередил время, и потребовалось 15 лет, чтобы этот замечательный язык завоевал популярность у ведущих разработчиков.

До официального переименования в "JavaScript", в выпуске Netscape Navigator 1995 года язык назывался сначала "Mocha", а затем "LiveScript". Слово "Java" в названии "JavaScript" не было случайным, хотя и не было очевидным: кроме общей синтаксической родословной, JavaScript имеет больше общего с Self (основанный на прототипах язык, разработанный в Xerox PARC в середине 1980-х годов) и Scheme (язык, разработанный в 1970-х Гаем Стилом (Guy Steele) и Джеральдом Сассманом (Gerald Sussman) под сильным влиянием Lisp и ALGOL), чем с Java. Айк был знаком и с Self, и с Scheme и использовал некоторые из их передовых парадигм в разработке "JavaScript". Название "JavaScript" частично было маркетинговой попыткой примазаться к успеху языка Java, которого он достиг в то время[1].

В ноябре 1996 года компания Netscape объявила о передаче JavaScript ассоциации Ecma — приватной международной некоммерческой организации по стандартизации, оказывающей существенное влияние на технологии и отрасли связи. Ассоциация Ecma International опубликовала первое издание спецификации ECMA-26, ставшее основой JavaScript.

Отношения между спецификациями Ecma (определяющими язык ECMAScript) и JavaScript являются главным образом академическими. Технически JavaScript — это

[1] Айк рассказал об этом в интервью в 2014 году.

реализация ECMAScript, но практически "JavaScript" и "ECMAScript" можно считать равнозначными терминами.

Последняя главная версия ECMAScript, 5.1 (обычно называемая "ES5"), была опубликована в июне 2011 года. Устаревшие браузеры, не поддерживающие ECMAScript 5.1, утратили популярность, и можно смело сказать, что ECMAScript 5.1 является текущим общепринятым языком веба.

Язык ECMAScript 6 (ES6), являющийся предметом рассмотрения этой книги, опубликован Ecma International в июне 2015 года. Рабочим названием этой спецификации до публикации было "Harmony" (Гармония), и вы полнее можете услышать такое название ES6, как "Harmony", "ES6 Harmony", "ES6", "ES2015" и "ECMAScript 2015". В этой книге мы называем его просто "ES6".

ES6

Внимательный читатель мог бы задаться вопросом "Если текущим общепринятым языком веба является ES5, то почему эта книга о ES6?"

Спецификация ES6 представляет существенное усовершенствование языка JavaScript, и некоторые из главных недостатков спецификации ES5 были устранены в ES6. Я полагаю, что вы найдете язык ES6 намного более приятным и мощным в применении (и ES5 был бы весьма хорошим началом). Кроме того (благодаря транскомпиляторам), сегодня вы можете написать код ES6 и транскомпилировать его в код, "совместимый с вебом" ES5.

И наконец после публикации ES6 поддержка этой спецификации браузерами будет устойчиво расти, и в некий момент транскомпиляция больше не будет необходимой для доступа широкой аудитории (я не настолько глуп, чтобы делать прогноз (даже грубый) о том, когда именно это случится).

Но что абсолютно ясно, так это то, что за ES6 — будущее разработки JavaScript, и, инвестировав свое время в его изучение, вы будете готовы к будущему, хотя и с транскомпиляцией, препятствующей ныне переносимости кода.

Однако сегодня не каждый разработчик имеет роскошь писать код ES6. Вполне возможно, что вы работаете с очень большим объемом существующего базового кода ES5, который весьма дорого преобразовать в код ES6. Некоторые разработчики просто не пожелают приложить дополнительные усилия, необходимые для транскомпиляции.

За исключением главы 1, в этой книге рассматривается ES6, а не ES5. По возможности я буду указывать, где ES6 отличается от ES5, но не ожидайте построчного сравнения примеров кода или обширного обсуждения, где "путь ES5" будет лучше, чем "путь ES6". Если вы относитесь к той категории программистов, которые по любой причине вынуждены придерживаться спецификации ES5, то эта книга не для вас (хотя я и надеюсь, что вы вернетесь к ней когда-либо в будущем!).

Выбор спецификации ES6 был сделан редакцией обдуманно. Усовершенствования ES6 достаточно существенны, чтобы затруднить четкое и ясное изложение материала. Короче говоря, книга, которая попыталась бы рассмотреть и ES5, и ES6, навредила бы обеим темам.

Для кого предназначена эта книга

Эта книга предназначена, прежде всего, для читателей, уже обладающих некоторым опытом программирования (освоивших хотя бы вводный курс программирования или сетевые классы). Новичкам в программировании эта книга тоже будет полезна, однако будет не лишним дополнить ее вводным курсом или классом.

Те, кто уже обладают некоторым опытом программирования в JavaScript (особенно если это только ES5), найдут практически полное описание важнейших концепций этого языка. Программистам, переходящим на JavaScript с другого языка, содержимое этой книги также должно понравиться.

В этой книге предпринята попытка всесторонне рассмотреть возможности языка, связанные с ним инструментальные средства, методики и парадигмы, которые управляют современной разработкой на JavaScript. Поэтому в данную книгу включены как простой и понятный материал (переменные, контроль потока, функции), так и довольно сложный (асинхронное программирование, регулярные выражения). В зависимости от своего уровня подготовки, вы можете найти одни главы более сложными, чем другие: начинающий программист, без сомнения, должен будет повторно пройти часть материала.

Для кого не предназначена эта книга

Эта книга — не полный справочник по JavaScript или связанным с ним библиотекам. Сеть Mozilla Developer Network (MDN) представляет собой превосходный, полный, актуальный и бесплатный *сетевой справочник по JavaScript*, на который я ссылаюсь повсюду в этой книге. Если вы предпочитаете физическую книгу, то книга Дэвида Флэнагана (David Flanagan) *JavaScript. Подробное руководство* является весьма подходящей (хотя на момент написания этой книги в ней ES6 не рассматривалась).

Соглашения, принятые в этой книге

Здесь используются соглашения, общепринятые в компьютерной литературе.

- Новые термины в тексте выделяются *курсивом*. Чтобы обратить внимание читателя на отдельные фрагменты текста, также применяется *курсив*.
- Текст программ, функций, переменных, URL веб-страниц и другой код выделены моноширинным шрифтом.

- Все, что придется вводить с клавиатуры, выделено **полужирным моноширинным** шрифтом.
- Знакоместо в описаниях синтаксиса выделено *курсивом*. Это указывает на необходимость заменить знакоместо фактическим именем переменной, параметром или другим элементом, который должен находиться на этом месте: `BINDSIZE=(`*`максимальная ширина колонки`*`)*(`*`номер колонки`*`)`.
- Пункты меню и названия диалоговых окон представлены следующим образом: Menu Option (Пункт меню).

Текст некоторых абзацев выделен специальным стилем. Это примечания, советы и предостережения, которые помогут обратить внимание на наиболее важные моменты в изложении материала и избежать ошибок в работе.

Этот элемент содержит совет или рекомендацию.

Этот элемент содержит примечание.

Этот элемент содержит предупреждение или предостережение.

Благодарности

Возможность писать книги для O'Reilly является огромной честью, и я должен поблагодарить Саймона С. Лорент (Simon St. Laurent) за то, что он увидел во мне потенциал и взял на борт. Мег Фоли (Meg Foley), мой редактор, была благосклонна, профессиональна и весьма полезна. Книга O'Reilly — это результат командных усилий, и мои литературный редактор Рейчел Монахан (Rachel Monaghan), выпускающий редактор Кристен Браун (Kristen Brown) и корректор Жасмин Куитин (Jasmine Kwityn) были быстры, обстоятельны и проницательны. Благодарю вас за все усилия!

Благодарю моих технических рецензентов Мэтт Инман (Matt Inman), Шелли Пауэрс (Shelley Powers), Ник Пинкхам (Nick Pinkham) и Коди Линдли (Cody Lindley) за содержательные отзывы, блестящие идеи и помощь в улучшении этой книги. Можно сказать, что без вас, возможно, неточностей было бы намного больше. Хотя все отзывы были невероятно полезны, я хочу выразить отдельную признательность Мэтту: его опыт педагога обеспечил ценную способность проникновения в суть, а отзывы Стивена Кольбера (Stephen Colbert) помогли мне сохранить свое здравомыслие!

Шелли Пауэрс (автор предыдущих изданий этой книги) заслуживает особенной благодарности не только за то, что передала эту тему мне, но и за то, что написала компетентный отзыв и помогла сделать эту книгу лучше (а также за некоторые бурные обсуждения!).

Я хотел бы выразить признательность всем читателям моей предыдущей книги (*Web Development with Node and Express*). Если бы вы не покупали ту книгу (и не отзывались о ней положительно!), у меня, вероятно, не было бы возможности написать эту книгу. Особая благодарность читателям, которые уделили время, чтобы прислать мне отзывы и исправления: я узнал многое из ваших писем!

Спасибо всем сотрудникам агентства Engineering at Pop Art, где я имею честь работать. Вы — моя судьба. Ваша поддержка важна для меня, ваш энтузиазм мотивирует меня, а ваши профессионализм и преданность — это именно то, что по утрам поднимает меня с постели. Том Пол (Tom Paul) заслуживает моей особой благодарности: его незыблемые принципы, инновационные деловые идеи и исключительные лидерские качества вдохновили меня приложить все усилия. Благодарю Стива Розенбаума (Steve Rosenbaum) за то, что он основал Pop Art, выдержал все бури и успешно передал факел Тому. Поскольку я был занят написанием этой книги, Колвин Фриц-Мур (Colwyn Fritze-Moor) и Эрик Бучман (Eric Buchmann) работали дополнительное время, чтобы выполнить те работы, которые обычно делал я. Спасибо обоим! Благодарю Дилана Халлстрома (Dylan Hallstrom) за то, что он был образцом надежности. Благодарю Лиз Том (Liz Tom) и Сэма Виски (Sam Wilskey) за то, что они присоединились к коллективу Pop Art! Благодарю Кэрол Харди (Carole Hardy), Никки Броволда (Nikki Brovold), Дженнифер Эртц (Jennifer Erts), Рэнди Кинер (Randy Keener), Патрика By (Patrick Wu) и Лайзу Мелог (Lisa Melogue) за всю поддержку. Наконец спасибо всем моим предшественникам, у которых я учился, а именно: Тони Алферезу (Tony Alferez), Полу Инману (Paul Inman) и Делу Олдсу(Del Olds).

Мой энтузиазм по поводу этой книги (и темы языков программирования в особенности) был заложен доктором Дэном Реслером (Dan Resler), адъюнкт-профессором Университета содружества Вирджинии. Я записался на его курс по теории компиляторов с абсолютным отсутствием интереса, а закончил его со страстью к теории формальных языков. Спасибо, что поделились своим энтузиазмом (и некой частью своей глубины понимания).

Благодарю всех моих друзей по PSU в когорте MBA. Это такое удовольствие — узнать вас всех! Особенная благодарность — Кэти, Аманде, Миске, Сахар, Полу С., Кэти, Джону Р., Лауре, Джоэл, Тайлер П., Тайлер С. и Джес: вы все обогатили мою жизнь!

Меня мотивируют не только сотрудники по Pop Art, но и мои друзья. Марк Бут (Mark Booth): никто из друзей не знает меня лучше; я доверил бы ему самые потаенные секреты. Твои творческий потенциал и талант вдохновляют меня. Надеюсь не разочаровать вас этой глупой книгой. Кэти Робертс (Katy Roberts) так же надежна,

как прилив, и так же красива. Кэти, благодарю вас за сердечную доброту и дружбу. Сара Льюис (Sarah Lewis): мне нравится ваше лицо. Байрон (Byron) и Эмбер Клэйтон (Amber Clayton) — истинные и верные друзья, которые всегда вызывают у меня улыбку. Лоррэйн (Lorraine), прошли годы, но ты находишь все самое лучшее во мне[2]. Кэйт Нахас (Kate Nahas): я был очень рад встретиться через столько лет; я надеюсь поднять тост в память Дьюка. Десембер: спасибо за ваше доверие, теплоту и дружеское отношение. Наконец благодарю моих новых друзей Криса Онстада (Chris Onstad) и Джессику Роу (Jessica Rowe): за прошлые два года вы двое принесли в мою жизнь так много радости и смеха, что я не знаю, как обходился бы без вас.

Моей матери, Энн: спасибо за вашу поддержку, любовь и терпение. Спасибо моему отцу, Тому, который остается для меня образцом любознательности, новаторства и самоотдачи. Без него я был бы плохим инженером (или, возможно, не был бы им вообще). Моя сестра, Мериес, всегда будет точкой опоры в моей жизни, демонстрируя преданность и уверенность.

[2] Слова из песни. — *Примеч. ред.*

От издательства

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать авторам.

Мы ждем ваших комментариев. Вы можете прислать письмо по электронной почте или просто посетить наш веб-сайт, оставив на нем свои замечания. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более подходящими для вас.

Отсылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш e-mail. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию следующих книг. Наши координаты:

E-mail: `info@dialektika.com`
WWW: `http://www.dialektika.com`

Наши почтовые адреса:

в России: 195027, Санкт-Петербург, Магнитогорская ул., д. 30, ящик 116
в Украине: 03150, Киев, а/я 152

Ваше первое приложение

Лучше всего учиться на *практике*. Именно поэтому мы начнем с создания простого приложения. Задача этой главы не в объяснении всего, что с этим связано: здесь слишком много незнакомого и неясного. Мой вам совет — расслабьтесь и не пытайтесь понять абсолютно все прямо сейчас. Эта глава должна заинтересовать вас. Просто наслаждайтесь поездкой, и к тому времени, когда вы закончите эту книгу, все в этой главе будет иметь для вас абсолютный смысл.

Если у вас нет серьезной практики в программировании, то одной из вероятных сложностей для вас будет поначалу то, как компьютеры воспринимают *литералы*. Человеческий разум способен справиться с неправильным текстом довольно легко, но не компьютеры. Если я сделаю грамматическую ошибку, то это может изменить ваше мнение о моей литературной грамотности, но вы, вероятно, все же поймете меня. У JavaScript, как и у всех языков программирования, нет возможности справляться с неправильным вводом. Регистр букв, орфография, порядок слов и пунктуация крайне важны. Если возникают проблемы, удостоверьтесь, что скопировали все правильно: возможно, вы не заметили точки с запятой, двоеточия, запятой или пробела, а возможно, вы смешали одиночные и парные кавычки или употребили прописные буквы вместо строчных. Приобретя немного опыта, вы узнаете, где можно "делать по-своему", а где следует быть совершенно дотошным. Пока же вам имеет смысл вводить код примеров точно так, как он написан.

По традиции книги по программированию начинаются с примера "Hello, World" (Привет, мир), который просто выводит на терминал фразу "hello world". Если интересно, то эту традицию заложил в 1972 году Брайан Керниган, специалист по информатике, работавший в Bell Labs. В печати это впервые появилось в книге *The C Programming Language*[1], опубликованной Брайаном Керниганом и Деннисом Ритчи

[1] Брайан У. Керниган, Деннис М. Ритчи. *Язык программирования C, 2-е издание*, ISBN 978-5-8459-1975-5, пер. с англ., ИД "Вильямс", 2017.

в 1978 году. Эта книга и по сей день является одной из наилучших и влиятельных книг по языкам программирования, и я почерпнул из нее немало вдохновения, работая над этой книгой.

Хотя "Hello, World" может показаться устаревшей традицией для современных поколений обучающихся программированию, скрытый смысл этой простой фразы остается сегодня таким же действенным, как и в 1978 году: это первые слова, произносимые кем-то, в кого *вы* вдохнули жизнь. Это свидетельство того, что вы — как Прометей, похитивший огонь у богов; как раввин, написавший истинное имя Бога на глине Голема; как доктор Франкенштейн, вдохнувший жизнь в свое создание[2]. Такое подобие творения, генезиса, и подвигло меня изначально к программированию. Возможно, однажды некий программист (может быть, и вы) даст жизнь первому искусственно разумному существу, и, возможно, его первыми словами будут "привет, мир".

В этой главе мы сбалансируем традицию, заложенную Брайаном Керниганом 44 года назад, искушенностью, доступной нынешним программистам. Мы увидим "hello world" на экране, но это будет далеко от тех примитивных слов, высветившихся пылающим фосфором на экране, которыми вы наслаждались бы в 1972 году.

С чего начать

В этой книге мы будем рассматривать использование JavaScript во всех его текущих воплощениях (сервер, сценарий, рабочий стол, для браузера и т.д.), но по историческим и практическим причинам мы собираемся начать с программы для браузера.

Одна из причин, по которым мы начинаем с примера, выполняемого в браузере, в том, что это дает нам свободный доступ к графическим библиотекам. Люди воспринимают информацию визуально, поэтому возможность соотнести концепции программирования с визуальными элементами — это весьма мощный инструмент обучения. В этой книге мы и так проведем много времени, уставившись на строки текста, но давайте начнем с чего-то немного более наглядного. Этот пример я также выбрал потому, что он органически знакомит с некоторыми очень важными концепциями, такими как управляемое событиями программирование, которое вам весьма пригодится в последующих главах.

Инструменты

Подобно тому, как без пилы у столяра не получится стол, мы не сможем написать программное обеспечение без некоторых инструментов. К счастью, необходимые в этой главе инструментальные средства минимальны: браузер и текстовый редактор.

[2] Надеюсь, что у вас будет больше сострадания к своим созданиям, чем у доктора Франкенштейна, и дела пойдут лучше.

Я счастлив сообщить, что на момент написания книги на рынке нет ни одного браузера, который не подходил бы для наших задач. Даже Internet Explorer, который долго был камнем в ботинке программистов, взялся за ум и стал теперь на равнее с Chrome, Firefox, Safari и Opera. Как уже говорилось, мой выбор — браузер Firefox, и здесь я буду описывать его особенности, которые помогут вам в работе. У других браузеров также есть эти возможности, но я опишу их так, как они реализуются в Firefox. Таким образом, при чтении этой книги вам имеет смысл использовать Firefox.

Вам понадобится текстовый редактор, чтобы писать код. Выбор текстовых редакторов может быть очень спорным (почти религиозные дебаты). В общем, текстовые редакторы можно подразделить на редакторы текстового режима и оконные редакторы. Два самых популярных редактора текстового режима — это vi/vim и Emacs. Одним из наибольших преимуществ редакторов текстового режима является то, что, кроме собственного компьютера, вы можете использовать их по SSH, т.е. вы можете соединиться с дистанционным компьютером и редактировать свои файлы в знакомом редакторе. Оконные редакторы выглядят современнее и обладают некоторыми полезными (и более знакомыми) элементами пользовательского интерфейса. В большинстве случаев, однако, вы будете редактировать только текст, поэтому оконный редактор не будет демонстрировать существенных преимуществ перед редактором текстового режима. Популярные оконные редакторы — это Atom, Sublime Text, Coda, Visual Studio, Notepad++, TextPad и Xcode. Если вы уже знакомы с одним из этих редакторов, то, вероятно, нет никакого резона его менять. Но если вы используете Блокнот из Windows, то я настоятельно рекомендую сменить его на более серьезный редактор (Notepad++ — простой и бесплатный выбор для пользователей Windows).

Описание всех возможностей вашего редактора выходит за рамки этой книги, но есть несколько средств, научиться использовать которые имеет смысл.

Выделение синтаксиса

Для выделения синтаксиса используются разные цвета, позволяющие различать синтаксические элементы в программе. Например, литералы могли бы быть одного цвета, а переменные — другого (что означают эти термины, вы узнаете вскоре!). Это может облегчить поиск проблем в коде. У большинства современных текстовых редакторов выделение синтаксиса есть и оно разрешено стандартно; если ваш код не разноцветный, обратитесь к документации своего редактора и узнайте, как включить возможность выделения.

Соответствие скобок

В большинстве языков программирования интенсивно используются круглые, фигурные и квадратные скобки. Иногда содержимое этих скобок охватывает много строк или даже несколько экранов; у вас будут скобки в пределах скобок, зачастую разных типов. Критически важно, чтобы количество открывающих скобок совпадало с количеством закрывающих, т.е. был их “баланс”; если этого нет, то ваша программа

не будет работать правильно. Редактор предоставляет визуальные маркеры в местах, где скобки открываются и закрываются, помогая вам выявить проблемы незакрытых скобок. Средства соответствия скобок в различных редакторах реализованы по-разному, от почти незаметного маркера до вполне очевидного. Несогласованные скобки — весьма распространенная причина ошибок у новичков, поэтому я настоятельно рекомендую узнать, как задействовать это средство в вашем редакторе.

Свертывание кода

Свертывание кода (code folding) несколько напоминает средство соответствия скобок. Свертывание кода позволяет временно скрыть часть кода, который вам не нужен в настоящее время. Термин происходит от идеи свертывания листа бумаги, чтобы скрыть незначительные детали. Как и соответствие скобок, свертывание кода по-разному реализовано в разных редакторах.

Автозавершение

Автозавершение (или *завершение слов* (word completion), или *IntelliSense*[3]) — весьма удобное средство, пытающееся предположить то, что вы вводите, прежде, чем закончите ввод. У этого средства две задачи. Первая — сэкономить время ввода. Вместо, например, слова `encodeURIComponent`, вы можете просто ввести `enc`, а затем выбрать encodeURIComponent из списка. Вторая задача — исследование. Например, если вы введете `enc`, потому что хотите ввести `encodeURIComponent`, то обнаружите, что есть еще функция `encodeURI`. В зависимости от редактора вы можете даже увидеть некоторую документацию, чтобы сделать выбор. Реализовать автозавершение для JavaScript сложнее, чем для многих других языков, поскольку это язык со слабой типизацией, а также из-за его правил областей видимости (о которых вы узнаете позже). Если автозавершение важно для вас, то, вероятно, придется присмотреться к ценам на редактор, удовлетворяющий вашим запросам. Здесь одни редакторы входят в состав пакета, а другие (vim, например) обеспечивают очень мощное автозавершение, но не без некоторой дополнительной настройки.

Комментарий о комментариях

В JavaScript, как и в большинстве языков программирования, есть синтаксис для *комментариев* (comment) в коде. Комментарии полностью игнорируются JavaScript; они предназначены только для вас и других программистов. Они позволяют добавлять в код объяснения, когда происходящее не ясно. В этой книге мы будем щедро использовать комментарии в примерах кода, чтобы объяснить происходящее.

В JavaScript есть два вида комментариев: встраиваемые и блоковые. Встраиваемые начинаются с двух косых черт (`//`) и простираются до конца строки. Блоковые

[3] Терминология Microsoft.

комментарии начинаются с косой черты и звездочки (/*), а завершаются звездочкой и косой чертой (*/). Они могут охватить несколько строк. Следующий пример демонстрирует оба типа комментариев.

```
console.log("echo"); // Выводит "echo" на консоль
/*
    Все в предыдущей строке, до пары косых черт, - это код JavaScript,
    подчиняющийся синтаксическим правилам. Две косые черты начинают
    комментарий, игнорируемый JavaScript. Этот текст находится в блоке
    комментариев и также будет проигнорирован JavaScript. Мы решили
    сделать отступ в этом блоке комментариев только для удобочитаемости,
    необходимости в нем нет.
*/
/* Смотри, мама, никакого отступа! */
```

Каскадные таблицы стилей (Cascading Style Sheet — CSS), которые мы рассмотрим вскоре, также используют синтаксис JavaScript для блоковых комментариев (встраиваемые комментарии в CSS не поддерживаются). В HTML (как и в CSS) нет встраиваемых комментариев, а его блоковые комментарии отличаются от JavaScript. Они окружены символами <!- и ->.

```
<head>
    <title>HTML and CSS Example</title>
    <!-- Это комментарий HTML...
         способный охватывать несколько строк. -->
    <style>
        body: { color: red; }
        /* Это комментарий CSS...
           способный охватывать несколько строк. */
    </style>
    <script>
        console.log("echo"); // Назад в JavaScript...
        /* ... где поддерживаются и встраиваемые,
            и блоковые комментарии. */
    </script>
</head>
```

Первые шаги

Мы собираемся начать с создания трех файлов: файла HTML, файла CSS и файла исходного кода JavaScript. Мы могли бы все сделать в файле HTML (код JavaScript и CSS может быть встроен в HTML), но в их раздельном хранении есть определенные преимущества. Если вы новичок в программировании, я настоятельно рекомендую следовать этим инструкциям шаг за шагом: в этой главе мы собираемся применить весьма любопытный подход, который облегчит процесс обучения.

Может показаться, что мы делаем слишком много работы для осуществления чего-то весьма простого, и некоторая правда в этом есть. Конечно, я мог бы предоставить пример, получающий тот же результат за значительно меньше этапов, но этим я *привил бы вам плохие привычки*. Представленные здесь дополнительные этапы вы будете видеть еще много раз, и хотя сейчас это может показаться сверхсложным, вы можете по крайней мере утешить себя тем, что учитесь делать все *правильно*.

Последнее важное примечание об этой главе. Это единственная глава в книге, в которой примеры кода будут написаны в синтаксисе ES5, а не ES6 (Harmony). Это сделано для гарантии того, что примеры кода будут выполняться, даже если вы не используете браузер, поддерживающий ES6. В следующих главах будет изложено, как написать код в ES6 и "транскомпилировать" его так, чтобы он выполнялся на устаревших браузерах. После рассмотрения этих основ в остальной части книги будет использован синтаксис ES6. Примеры кода в этой главе достаточно просты, и использование ES5 не представляет существенного препятствия.

Файлы, создаваемые для этого упражнения, должны находиться в той же папке. Я рекомендую создать для этого примера новую папку, чтобы файлы не потерялись среди других.

Начнем с файла JavaScript. Используя текстовый редактор, создайте файл `main.js`. Добавьте в него одну следующую строку.

```
console.log('main.js loaded');
```

Затем создайте файл CSS, `main.css`. Пока у нас нет ничего, что стоило бы вставить здесь, поэтому включим в него только комментарий, чтобы не было пустого файла.

```
/* Здесь будут стили. */
```

Затем создайте файл с именем `index.html`.

```
<!doctype html>
<html>
    <head>
        <link rel="stylesheet" href="main.css">
    </head>
    <body>
        <h1>My first application!</h1>
        <p>Welcome to <i>Learning JavaScript, 3rd Edition</i>.</p>

        <script src="main.js"></script>
    </body>
</html>
```

Хотя эта книга и не об HTML или разработке веб-приложений, большинство из вас изучает JavaScript именно с этой целью, поэтому мы затронем некоторое

количество аспектов языка HTML, поскольку они касаются разработки на JavaScript. Документ HTML состоит из двух основных частей: *заголовка* (head) и *тела* (body). В заголовке содержится информация, которая *непосредственно не отображается в браузере* (хотя она может повлиять на то, что отображается в браузере). В теле находится содержимое страницы, которая будет отображена в браузере. Важно понимать, что элементы в заголовке никогда не будут представлены в браузере, тогда как элементы в теле обычно отображаются (некоторые типы элементов, такие как `<script>`, не будут видимы, и стили CSS также способны скрыть элементы тела).

В заголовке содержится строка `<link rel="stylesheet" href="main.css">`; вот так пустой в настоящее время файл CSS связывается с вашим документом. Затем, в конце тела, имеется строка `<script src="main.js"></script>`, связывающая файл JavaScript с вашим документом. Может показаться странным, что один подключается в заголовке, а другой в конце тела. Дескриптор `<script>` можно, конечно, поместить и в заголовок, однако по некоторым причинам, включая производительность, имеет смысл помещать его в конец тела.

В теле имеется дескриптор `<h1>My first application!</h1>`, представляющий собой текст заголовка первого уровня (означающий самый больший и важный текст на странице), сопровождаемый дескриптором `<p>` (параграф), содержащим некий текст, часть которого выделена курсивом (дескриптор `<i>`).

Найдите и загрузите файл `index.html` в свой браузер. В большинстве систем проще всего сделать это, дважды щелкнув на файле в средстве просмотра файлов (можно также перетащить файл в окно браузера). Вы увидите содержимое тела своего файла HTML.

В этой книге много примеров кода. Поскольку файлы HTML и JavaScript могут стать очень большими, я не буду представлять их содержимое целиком каждый раз: вместо этого я объясню в тексте, где фрагмент кода располагается в файле. У начинающих программистов это может вызвать некоторое неудобство, однако понять способ сборки кода необходимо и очень важно.

Консоль JavaScript

Мы уже написали некий код JavaScript: `console.log('main.js loaded')`. Что он делает? *Консоль* (console) — это текстовый инструмент для программистов, помогающий им диагностировать свою работу. Вы будете часто использовать консоль по мере изучения этой книги.

Разные браузеры предоставляют разные способы обращения к консоли. Поскольку вы будете делать это весьма часто, я рекомендую узнать соответствующую комбинацию клавиш. В Firefox — это <Ctrl+Shift+K> (для Windows и Linux) или <Command+Option+K> (для Mac).

В окне, в котором загружен файл `index.html`, откройте консоль JavaScript; вы должны увидеть текст "main.js loaded" (main.js загружен) (если вы не видите его, попробуйте перезагрузить страницу). `console.log` — это метод[4] вывода на консоль, весьма полезный при отладке и подобном изучении.

Одним из многих преимуществ консоли является возможность, кроме наблюдения вывода своей программы, *непосредственно вводить код JavaScript*, проверяя что-то таким образом, изучая возможности JavaScript или даже внося временные изменения в свою программу.

Библиотека jQuery

Мы собираемся добавить к нашей странице чрезвычайно популярную клиентскую библиотеку сценариев — *jQuery*. Хотя это и не обязательно, а для данной простой задачи даже избыточно, именно такая вездесущая библиотека зачастую является первой включаемой в код веба. Даже при том, что в этом примере мы могли бы легко обойтись без библиотеки jQuery, вскоре вы привыкнете встречать ее в своем коде.

Библиотеку jQuery мы подключаем в конце тела *перед* собственно файлом `main.js`:

```
<script src="https://code.jquery.com/jquery-2.1.1.min.js"></script>

<script src="main.js"></script>
```

Обратите внимание, что мы используем адрес URL из Интернета, а это значит, что без доступа к Интернету ваша страница не будет работать правильно. Мы подключаем библиотеку jQuery из открытой *сети доставки контента* (Content Delivery Network — CDN), обладающей определенными преимуществами по производительности. Если вы будете работать над своим проектом без подключения к сети, придется загрузить файл и подключать его со своего компьютера. Теперь мы изменим свой файл `main.js` так, чтобы использовать в своих интересах одно из средств jQuery:

```
$(document).ready(function() {
    'use strict';
    console.log('main.js loaded');
});
```

Если у вас еще нет опыта использования библиотеки jQuery, то это, вероятно, выглядит непонятно. Здесь будет много такого, что станет понятным намного позже. В данном случае jQuery позволяет удостовериться, что браузер загрузил весь код HTML, прежде чем выполнить наш код JavaScript (который в настоящее время состоит только из одной команды `console.log`). Всякий раз, работая с кодом JavaScript в браузере, мы будем делать это только для практики: любой код JavaScript, который

[4] Более подробная информация о различии между *функцией* и *методом* приведена в главе 9.

вы пишете, располагается между строками `$(document).ready(function() {` и `});`. Обратите также внимание на то, что строка `'use strict'` — это нечто, о чем вы узнаете больше попозже, но в основном она указывает интерпретатору JavaScript обрабатывать ваш код более жестко. Хотя сначала это может показаться не очень хорошей идеей, фактически это помогает писать лучший код JavaScript и предотвращает наиболее распространенные ошибки. В этой книге мы, конечно, будем учиться писать очень строгий код JavaScript!

Рисование графических примитивов

Одним из множества преимуществ HTML5 является стандартизированный графический интерфейс. *Холст* (canvas) HTML5 позволяет рисовать такие графические примитивы, как квадраты, круги и многоугольники. Непосредственное использование холста может быть затруднительно, поэтому мы будем применять графическую библиотеку `Paper.js`, чтобы использовать в своих интересах холст HTML5.

`Paper.js` — не единственная доступная графическая библиотека. Весьма популярны и надежны такие альтернативы, как `KineticJS`, `Fabric.js` и `EaselJS`. Я использовал все эти библиотеки, и все они очень высокого качества.

Прежде чем мы начнем использовать библиотеку `Paper.js`, нам понадобится элемент холста HTML для рисования. Добавьте в тело следующее (можно поместить куда угодно, например после вводного параграфа):

```
<canvas id="mainCanvas"></canvas>
```

Обратите внимание, что мы присвоили холсту атрибут `id`: так нам будет легче обращаться к нему из кода JavaScript и CSS. Если мы загрузим свою страницу прямо сейчас, то не увидим никаких различий; мало того что мы ничего не получили на холсте, это белый холст на белом листе, не имеющий ни ширины, ни высоты. Его действительно очень трудно увидеть.

У каждого элемента HTML может быть идентификатор. Чтобы быть допустимым (правильным), каждый идентификатор должен быть уникален. Создав холст с идентификатором "`mainCanvas`", мы не можем повторно использовать этот идентификатор. Поэтому рекомендуется экономно использовать идентификаторы. Мы используем этот идентификатор здесь потому, что новичкам зачастую проще знакомиться с одной вещью за раз и по определению идентификатор может относиться только к одной вещи на странице.

Давайте изменим файл `main.css` так, чтобы наш холст выделялся на странице. Если вы не знакомы с CSS, то это нормально. CSS просто устанавливает ширину и высоту для нашего элемента HTML, а также добавляет черную границу.[5]

```
#mainCanvas {
    width: 400px;
    height: 400px;
    border: solid 1px black;
}
```

Если вы перезагрузите свою страницу, то увидите холст.

Получив холст для рисования, давайте подключим библиотеку `Paper.js`, чтобы она помогла нам с рисунком. Сразу после подключения библиотеки jQuery, но до подключения собственного файла `main.js`, добавьте следующую строку.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/paper.js/0.9.24/ ¿
paper-full.min.js"></script>
```

Обратите внимание: для подключения библиотеки `Paper.js` в наш проект мы используем CDN, как и в случае с библиотекой jQuery.

Вы уже начали понимать, что порядок подключения очень важен. Поскольку мы собираемся использовать библиотеки jQuery и `Paper.js` в нашем файле `main.js`, обе они подключаются первыми. Ни одна из них не зависит от другой, поэтому не имеет значения, какая из библиотек подключается первой, но я всегда подключаю первой библиотеку jQuery (в силу привычки), поскольку очень многое в веб-разработке зависит от нее.

Теперь, подключив библиотеку `Paper.js`, проделаем небольшую работу по ее настройке. Подобный часто встречаемый код (повторяющийся перед вашим собственным) зачастую называют *шаблоном* (boilerplate). Добавьте следующее в файл `main.js`, сразу после `'use strict'` (если хотите, можете удалить `console.log`):

```
paper.install(window);
paper.setup(document.getElementById('mainCanvas'));

// TODO

paper.view.draw();
```

В первой строке библиотека `Paper.js` устанавливается в глобальную область видимости (что будет иметь больше смысла в главе 7). Во второй строке библиотека `Paper.js` подключается к холсту и готовится к рисованию. В середине, где мы

[5] Всем, кто желает узнать больше о CSS и HTML, я рекомендую бесплатный курс по HTML и CSS на Codecademy.

поместили комментарий TODO, будет расположен фактически интересный материал. В последней строке Paper.js получает инструкцию нарисовать нечто на экране.

Теперь, когда с шаблоном покончено, давайте что-нибудь нарисуем! Начнем с зеленого круга в середине холста. Замените комментарий "TODO" следующими строками.

```
var c = Shape.Circle(200, 200, 50);
c.fillColor = 'green';
```

Обновите свой браузер и полюбуйтесь зеленым кругом. Вы написали свой первый реальный код JavaScript. Фактически в этих двух строках происходит довольно много, но пока важно знать лишь несколько вещей. В первой строке создается *объект* (object) круга с использованием трех *аргументов* (argument): координат *x* и *y* центра круга, а также его радиуса. Помните, мы создали свой холст размером 400 пикселей шириной и 400 пикселей высотой, поэтому центр холста находится в точке с координатами (200, 200). И радиус 50 делает круг размером в одну восьмую ширины и высоты холста. Во второй строке устанавливается *цвет заполнения*, отличный от цвета контура (*штрих* (stroke), на языке Paper.js). Не бойтесь экспериментировать с изменением этих аргументов.

Автоматизация повторяющихся задач

Предположим, необходимо не просто добавить один круг, а заполнить холст ими, расположив в табличном порядке. Если сделать круги немного меньше, то на холсте можно разместить 64 круга. Конечно, вы могли бы скопировать только что написанный код 63 раза и вручную модифицировать все координаты так, чтобы круги располагались в виде таблицы. Похоже на большое количество работы, не так ли? К счастью, компьютер отлично подходит для повторяющихся задач такого вида. Давайте рассмотрим, как мы можем нарисовать 64 равномерно расположенных круга. Заменим свой код, рисующий одиночный круг, следующим.

```
var c;
for(var x=25; x<400; x+=50) {
    for(var y=25; y<400; y+=50) {
        c = Shape.Circle(x, y, 20);
    c.fillColor = 'green';
    }
}
```

Если вы обновите свой браузер, то увидите 64 зеленых круга! Если вы новичок в программировании, то только что написанное может показаться непонятным, однако вполне очевидно, что это лучше написания 128 строк вручную.

То, что мы использовали, называется циклом for. Он является частью синтаксиса управления потоком, о котором вы узнаете подробно из главы 4. Цикл for позволяет

задать начальное условие (25), условие завершения (меньше, чем 400) и значение инкремента (50). Мы используем один цикл внутри другого для выполнения задачи по осям x и y.

Этот пример можно было бы написать многими способами. В данном случае мы сделали координаты `x` и `y` важной частью информации: мы явно определяем, где круги начинаются и как далеко расположены один от другого. К решению этой задачи мы могли бы подойти и с другой стороны: как упоминалось, есть необходимое количество кругов (64), и программа может сама выяснить, как их распределить, чтобы они поместились на холсте. Причина выбора нашего решения в том, что оно лучше соответствует альтернативе копированию и вставке своего кода создания круга 64 раза и самостоятельного задания интервалов.

Обработка пользовательского ввода

Все, что мы делали до сих пор, не требовало ввода от пользователя. Пользователь может щелкать на кругах, но это ничего не даст. Аналогично попытка перетащить круг не даст никакого эффекта. Давайте немного улучшим интерактивность, разрешив *пользователю* выбирать, где будут нарисованы круги.

Важно привыкнуть к *асинхронному* характеру пользовательского ввода. *Асинхронное событие* (asynchronous event) — это событие, момент наступления которого вы не можете никак контролировать. Щелчок мышью пользователя — это пример асинхронного события: вы не можете знать, когда пользователь соберется щелкнуть. Конечно, вы можете попросить его щелкнуть, но когда пользователь щелкнет (и щелкнет ли вообще) — это его и только его дело. Асинхронные события, являющиеся результатом пользовательского ввода, имеют интуитивно понятный смысл, но мы рассмотрим намного менее интуитивно понятные асинхронные события в последующих главах.

Для обработки пользовательского ввода библиотека `Paper.js` использует объект под названием "`t●ol`" (инструмент). Если такой выбор имени не кажется вам интуитивно понятным, то вы находитесь в хорошей компании: я тоже не знаю, почему разработчики `Paper.js` использовали эту терминологию.[6] Возможно, вам лучше рассматривать "tool" (инструмент) как "user input tool" (инструмент пользовательского ввода). Давайте заменим наш код, рисовавший таблицу кругов, следующим кодом:

```
var tool = new Tool();

tool.onMouseDown = function(event) {
    var c = Shape.Circle(event.point.x, event.point.y, 20);
```

[6] Технический рецензент Мэтт Инман предположил, что разработчики `Paper.js`, возможно, были пользователями Photoshop, знакомыми с инструментами "hand tool", "direct selection tool" и т.д.

```
    c.fillColor = 'green';
};
```

На первом этапе этот код создает наш объект инструмента. Сделав это, мы можем закрепить за ним *обработчик событий* (event handler), в данном случае — обработчик событий по имени `onMouseDown`. Всякий раз, когда пользователь щелкает мышью, *вызывается функция, которую мы закрепили за этим обработчиком.* Это очень важно понять. Наш предыдущий код выполнялся сразу же: мы обновили браузер, и зеленые круги появились автоматически. Здесь этого не происходит, а если бы произошло, то был бы нарисован один зеленый круг где-нибудь на экране. Вместо этого код, содержавшийся между фигурными скобками после слова `function`, выполняется *только тогда, когда пользователь щелкает мышью на холсте.*

Обработчик событий делает две вещи: выполняет ваш код, *когда* происходит щелчок мышью, и указывает вам, *где* произошел щелчок. Место щелчка хранится в свойстве аргумента, `event.point`, у которого есть два свойства, `x` и `y`, указывающих координаты щелчка.

Обратите внимание: мы могли бы сэкономить на вводе кода, непосредственно передав точку в `Circle` (вместо передачи координат x и y по отдельности).

```
var c = Shape.Circle(event.point, 20);
```

Это подчеркивает очень важный аспект JavaScript: он в состоянии узнавать информацию о передаваемых переменных. В предыдущем случае, встретив три числа подряд, он знает, что они представляют координаты *x*, *y* и радиус. Если он встречает два аргумента, то знает, что первый — объект типа `point`, а второй — радиус. Больше об этом мы узнаем в главах 6 и 9.

Программа Hello, World

Давайте завершим эту главу примером Брайана Кернигана 1972 года. Вся тяжелая работа уже сделана, осталось лишь добавить текст. Перед вашим обработчиком `onMouseDown` добавьте следующее.

```
var c = Shape.Circle(200, 200, 80);
c.fillColor = 'black';
var text = new PointText(200, 200);
text.justification = 'center';
text.fillColor = 'white';
text.fontSize = 20;
text.content = 'hello world';
```

Это дополнение довольно простое: мы создаем другой круг, который будет фоном для нашего текста, а затем фактически создаем текстовый объект (`PointText`). Мы определяем место вывода (центр экрана) и несколько дополнительных свойств

(выравнивание, цвет и размер). Наконец мы определяем фактическое текстовое содержимое ("hello world" (привет мир)).

Обратите внимание на то, что это не первый случай, когда мы выводим текст в JavaScript: мы сделали это с `console.log` ранее в данной главе. Конечно, мы могли изменить этот текст на "hello world". Так или иначе, это больше походило бы на опыт, который вы получили бы в 1972 году, но задача примера — не текст или его визуализация: дело в том, что вы создаете нечто автономное, имеющее зримый эффект.

Обновив свой браузер с этим кодом, вы приняли участие в почтенной традиции примеров "Hello, World". Если это ваше первое приложение "Hello, World", то поздравляю: вы приняты в клуб! Если нет, то я надеюсь, что этот пример дал вам некое понимание JavaScript.

Инструменты разработки JavaScript

Хотя для написания кода JavaScript вполне достаточно текстового редактора и браузера (как мы видели в предыдущей главе), разработчики JavaScript используют некоторые весьма полезные инструменты разработки. Кроме того, поскольку мы сосредоточиваемся на спецификации ES6, в остальной части этой книги нам понадобится способ преобразования нашего кода ES6 в переносимый код ES5. Обсуждаемые в этой главе инструментальные средства весьма популярны, и вы, вероятно, встретитесь с ними в любом проекте с открытым исходным кодом или в группе разработки программного обеспечения. К ним относятся следующие.

- Git — инструмент контроля версий, помогающий управлять проектом по мере его роста и организовывать взаимодействие с другими разработчиками.
- Node позволяет запускать код JavaScript вне браузера (поставляется в комплекте с npm, обеспечивающим доступ к остальной части инструментальных средств этого списка).
- Gulp — *инструмент сборки* (build tool), автоматизирующий общие задачи разработки (популярная альтернатива — Grunt).
- Babel — *транскомпилятор* (transcompiler), преобразующий код ES6 в переносимый код ES5.
- ESLint — *анализатор* (linter), помогающий избежать наиболее распространенных ошибок и делающий вас лучшим программистом!

Не считайте эту главу отклонением от основной темы (JavaScript). Рассматривайте ее как практическое введение в некие важные инструментальные средства и методики, общепринятые в разработке JavaScript.

Написание кода ES6 сегодня

У меня есть хорошие новости и плохие новости. Хорошие новости — спецификация ES6 (или Harmony, или JavaScript 2015) является великолепным, восхитительным этапом развития в истории JavaScript. Плохие новости — мир еще не совсем готов

к этому. Это не означает, что вы не можете использовать его сейчас, просто на программиста возлагается дополнительная работа, поскольку код ES6 должен быть транскомпилирован в "безопасный" код ES5 для гарантии его выполнения везде.

Программисты с большим опытом могли бы подумать "Большое дело! Когда-то давно не было такой вещи, как язык, который не требовал бы компиляции!" Я пишу программное обеспечение достаточно давно, чтобы помнить то время, но я не восхищаюсь им: я наслаждаюсь отсутствием суеты в таких интерпретируемых языках, как JavaScript.[1]

Одним из преимуществ JavaScript всегда была его вездесущность: он стал стандартным языком сценариев браузеров почти внезапно, а с появлением Node его использование расширилось за пределы браузера. Таким образом, будет немного печально узнать, что несколько ближайших лет вы, вероятно, не сможете использовать код ES6, не заботясь о поддерживающих его браузерах. Если вы — разработчик Node, ситуация немного проще: поскольку у вас есть только один процессор JavaScript, вы можете проследить прогресс поддержки ES6 в Node.

Примеры кода ES6 из этой книги можно запустить в Firefox или на таком веб-сайте, как *ES6 Fiddle*. Однако для кода реальных проектов вам понадобятся инструменты и методики, описанные в этой главе.

Интересный аспект перехода JavaScript с ES5 на ES6 — в отличие от предыдущих выпусков, данный является достаточно *плавным*. Таким образом, у браузера, который вы используете прямо сейчас, вероятно, есть некоторые (но не все) возможности, доступные в ES6. Этот постепенный переход стал возможен частично благодаря динамической природе JavaScript, а частично благодаря изменчивой природе обновлений браузера. Возможно, вы слышали, что для описания браузеров используют термин *вечнозеленый* (evergreen): изготовители браузеров уходят от концепции отдельных версий, которые следует обновлять. Браузеры, как рассуждают они, должны быть в состоянии совершенствовать себя, поскольку они всегда подключаются к Интернету (по крайней мере, если они собираются быть полезными). У браузеров все еще есть версии, но теперь вполне резонно подразумевать, что у ваших пользователей есть *последняя* версия, поскольку вечнозеленые браузеры не позволяют пользователям *избежать* обновлений.

Но даже при вечнозеленых браузерах потребуется некоторое время, прежде чем вы сможете полагаться на доступность всего великолепия возможностей ES6 на стороне клиента. Поэтому в настоящее время *транскомпиляция* (transpilation) — это жизненно важный факт.

[1] Некоторые процессоры JavaScript (например, Node) компилируют код JavaScript, но это происходит неявно.

Возможности ES6

У спецификации ES6 так много новых возможностей, что даже транскомпиляторы, о которых мы будем говорить, в настоящее время поддерживают не все из них. Чтобы помочь контролировать хаос, разработчик *kangax* из Нью-Йорка предоставляет превосходную *таблицу совместимости* средств ES6 (и ES7). На август 2015 года даже наиболее полная реализация (Babel) поддерживала только 72%. К счастью, в Babel доступны самые важные возможности, которые были реализованы с самого начала, и все средства, обсуждаемые в этой книге.

Прежде чем мы сможем начать транскомпиляцию, необходимо проделать немного подготовительной работы. Следует удостовериться в наличии всех необходимых инструментов и научиться создавать новые проекты (пройдя процесс несколько раз, вы станете делать это автоматически). Впоследствии при запуске новых проектов вы, вероятно, захотите вернуться к этой главе.

Установка Git

Если на вашем компьютере Git еще не установлен, найдите и загрузите его (с инструкциями для вашей операционной системы) с *домашней страницы Git*.

Терминал

В этой главе мы будем работать в *терминале* (terminal), который также называется *командной строкой* (command line) или *командной оболочкой* (command shell)). Терминал — это текстовый способ взаимодействия с компьютером, общепринятый среди программистов. Хотя вполне можно, конечно, быть весьма успешным программистом и никогда не использовать терминал, я полагаю, что это важный навык: во многих учебниках и книгах подразумевается, что вы используете терминал, а множество инструментальных средств разработано так, чтобы могли использоваться на терминале.

Наиболее популярна терминальная оболочка (терминальный интерфейс) *bash*; она доступна изначально на машинах с Linux, и OS X. Несмотря на то что у Windows есть собственная оболочка командной строки, система Git (которую мы установим далее) предоставляет интерфейс командной строки bash, который я и рекомендую использовать. В этой книге мы будем использовать bash.

Запустив терминал, вы увидите *приглашение* (prompt); после него вы будете вводить команды. Стандартное приглашение может включать имя вашего компьютера или каталога, в котором вы находитесь, и заканчиваться знаком доллара (`$`). Таким образом, в примерах кода этой главы я буду использовать знак доллара, чтобы указать приглашение. За приглашением следует то, что вы должны ввести. Например, чтобы получить список файлов в текущем каталоге, введите после приглашения `ls`.

```
$ ls
```

В Unix, а следовательно, и в bash, имена каталогов разделяются символами прямой наклонной черты (/). В операционной системе Windows, в которой каталоги разделяются символами обратной наклонной черты (\), Git преобразует их в символы прямой наклонной черты. В качестве сокращения для вашего основного каталога (в котором вы обычно храните свои файлы) bash использует тильду (~).

Вам нужно уметь изменять текущий каталог (`cd`) и создавать новые каталоги (`mkdir`). Например, чтобы перейти в свой основной каталог, введите

```
$ cd ~
```

Команда `pwd` (print working directory — вывести рабочий каталог) указывает, в каком каталоге вы находитесь в настоящий момент.

```
$ pwd
```

Чтобы создать подкаталог `test`, введите

```
$ mkdir test
```

Чтобы перейти в этот вновь созданный каталог, введите

```
$ cd test
```

Две точки (`..`) — это сокращение для родительского каталога. Чтобы перейти "вверх" по каталогам (если вы следовали вперед, то это возвратит вас к основному каталогу), введите

```
$ cd ..
```

О терминале можно узнать еще много всего, но эти основные команды — все, что необходимо для изучения материала данной главы. Если вы хотите узнать больше, я рекомендую книгу Скотта Граннемана *Linux. Карманный справочник, 2-е издание* (пер. с англ., ИД "Вильямс" 2016, ISBN 978-5-8459-2101-7).

Корневой каталог проекта

Для каждого проекта желательно создать собственный каталог. Мы называем этот каталог *корневым каталогом проекта* (project root) или корнем. Например, если следовать примерам в этой книге, то можно создать каталог `lj`, который был бы корневым каталогом вашего проекта. Во всех примерах командной строки этой книги мы подразумеваем, что вы находитесь в корневом каталоге проекта. Если вы пытаетесь запустить пример, а он не работает, первое, что стоит проверить, — это находитесь ли вы в корневом каталоге проекта. Любые файлы, которые мы создаем, будут расположены относительно корневого каталога проекта. Например, если корневой каталог вашего проекта — `/home/joe/work/lj` и мы просим, чтобы вы создали файл

public/js/test.js, то полный путь к этому файлу должен быть /home/joe/work/lj/public/js/test.js.

Git: контроль версий

Мы не будем обсуждать контроль версий подробно в этой книге, но если вы еще его не используете, то стоит начать. Если вы не знакомы с Git, я рекомендую использовать эту книгу для возможности попрактиковаться.

Сначала из своего корневого каталога проекта инициализируйте хранилище.

```
$ git init
```

Это создаст хранилище проекта (теперь в вашем корневом каталоге проекта есть скрытый каталог по имени .git).

Обязательно будет несколько файлов, которые не нужно отслеживать при контроле версий: элементы сборки, временные файлы и т.п. Эти файлы могут быть явно исключены, если перечислить их в файле .gitignore. Поэтому создайте файл .gitignore со следующим содержимым.

```
# Отладочные журналы npm
npm-debug.log*

# Зависимости проекта
node_modules

# Атрибуты папки OSX
.DS_Store

# Временные файлы
*.tmp
*~
```

Если есть любые другие "вспомогательные" файлы, о которых вы знаете, то также добавьте их сюда (например, если вы знаете, что ваш редактор создает файлы .bak, то добавьте в этот список запись *.bak).

Команда git status, которую вы будете вводить довольно часто, указывает вам текущее состояние вашего хранилища. Если ввести ее сейчас, то можно будет увидеть следующее.

```
$ git status
On branch master

Initial commit

Untracked files:
    (use "git add <file>..." to include in what will be committed)
```

```
        .gitignore

nothing added to commit but untracked files present (use "git add" to track)
```

Git сообщает вам важную новость о том, что в каталоге есть новый файл (`.gitignore`), но он `untracked`, т.е. Git не отслеживает его.

Основная единица работы в хранилище Git — это *фиксация изменений* (commit). В настоящее время в вашем хранилище еще не зафиксировано никаких изменений (вы только что инициализировали его и создали файл, но ни одна команда фиксации Git выполнена не была). Git не делает предположений о том, что вы хотите отслеживать, поэтому вы должны явно добавить `.gitignore` в хранилище.

```
$ git add .gitignore
```

Мы все еще не создали фиксацию изменений; мы просто *организовали* файл `.gitignore` так, чтобы войти в следующую фиксацию изменений. Выполнив команду `git status` снова, мы увидим следующее.

```
$ git status
On branch master

Initial commit

Changes to be committed:
    (use "git rm --cached <file>..." to unstage)

        new file: .gitignore
```

Теперь изменения файла `.gitignore` должны быть *фиксируемыми* (committed). Мы все еще не создали фиксацию изменений, но когда мы это сделаем, наши изменения появятся в файле `.gitignore`. Мы могли бы добавить и больше файлов, но давайте сейчас создадим фиксацию изменений.

```
$ git commit -m "Initial commit: added .gitignore."
```

Строка, следующая за `-m`, является *сообщением* о фиксации изменений, кратко описывающим действия, осуществленные в данной фиксации изменений. Это позволяет оглядываться назад при фиксациях изменений и просматривать историю выполнения проекта.

Вы можете рассматривать фиксацию изменений как снимок своего проекта в определенный момент времени. Мы только что сделали снимок проекта (пока только с одним файлом `.gitignore`), и вы можете вернуться к нему в любое время. Если вы выполните команду `git status` теперь, то получите следующее.

```
On branch master
nothing to commit, working directory clean
```

Давайте внесем немного дополнительных изменений в наш проект. В файле `.gitignore` мы игнорируем любые файлы с именем `npm-debug.log`, но, скажем, мы

хотим игнорировать любые файлы с расширением `.log` (что является стандартной практикой). Отредактируйте файл `.gitignore` и измените эту строку на `*.log`. Давайте также добавим файл с именем `README.md`, который является стандартным файлом, объясняющим проект в популярном формате `Markdown`:

```
= Learning JavaScript, 3rd Edition
== Chapter 2: JavaScript Development Tools

In this chapter we're learning about Git and other
development tools.
```

Теперь введите **`git status`**.

```
$ git status
On branch master
Changes not staged for commit:
    (use "git add <file>..." to update what will be committed)
    (use "git checkout -- <file>..." to discard changes in working directory)

        modified: .gitignore

Untracked files:
    (use "git add <file>..." to include in what will be committed)

        README.md
```

У нас теперь есть два изменения: одно — в отслеживаемом файле (`.gitignore`) и одно — в новом файле (`README.md`). Мы могли бы добавить изменения так, как прежде.

```
$ git add .gitignore
$ git add README.md
```

Но на сей раз мы будем использовать сокращение, чтобы добавить *все* изменения, а затем создавать фиксацию изменений со всеми этими изменениями.

```
$ git add -A
$ git commit -m "Ignored all .log files and added README.md."
```

Это обычный шаблон, который вы будете часто повторять (добавление изменений и последующая фиксация изменений). Старайтесь делать свои фиксации изменений небольшими и логически единообразными, как будто рассказываете кому-то историю, объясняя ход своих мыслей. Всякий раз внося изменения в свое хранилище, вы будете следовать тому же шаблону: добавьте одно или несколько изменений, а затем создайте их фиксацию.

```
$ git add -A
$ git commit -m "<brief description of the changes you just made>"
```

Команда `git add` вызывает у новичков недопонимание; создается впечатление, что вы добавляете файлы в хранилище. Эти изменения *могут* быть новыми файлами, но это вполне может быть внесением изменений в файлы, уже находящиеся в хранилище. Другими словами, вы добавляете *изменения*, а не файлы (а новый файл — это только частный случай изменений).

Это самый простой рабочий цикл Git; если вы хотите узнать больше о Git, я рекомендую руководство *Git Tutorial* на GitHub и книгу *Version Control with Git, Second Edition* Джона Лолигера (Jon Loeliger) и Мэтью Маккалоу (Matthew McCullough).

Управление пакетами: npm

Для разработки кода на JavaScript знать npm не обязательно, но этот инструмент управления пакетом становится все более и более популярным. Фактически он необходим при разработке для Node. Пишете ли вы приложения для Node или только для браузера, вы найдете, что с npm жить намного проще. В частности, мы будем использовать npm для установки наших инструментов сборки и транскомпиляторов.

npm поставляется вместе с Node, поэтому, если вы еще не установили Node, перейдите на *домашнюю страницу Node.js* и щелкните на большой зеленой кнопке INSTALL (Установить). Как только вы установите Node, убедитесь, что npm и Node функционируют в вашей системе. В командной строке выполните следующее.

```
$ node -v
v4.2.2
$ npm -v
2.14.7
```

Ваши номера версий Node и npm могут быть другими. В широком смысле npm управляет установленными пакетами. Пакет может быть чем угодно, от всего приложения до фрагмента кода, модуля или библиотеки, которую вы будете использовать в своем проекте.

npm обеспечивает установку пакетов на двух уровнях: глобальном и локальном. Глобальные пакеты — это обычно инструментальные средства командной строки, которые вы будете использовать в процессе разработки. Локальные пакеты принадлежат данному проекту. Установка пакета осуществляется командой `npm install`. Давайте установим популярный пакет `Underscore`, чтобы увидеть, как он работает. В корневом каталоге своего проекта выполните следующее.

```
$ npm install underscore
underscore@1.8.3 node_modules\underscore
```

npm сообщает, что он установил последнюю версию Underscore (у меня на момент написания книги — 1.8.3; у вас, вероятно, будет другая). Underscore — это модуль без зависимостей, поэтому вывод npm очень краток; для некоторых сложных

модулей вы можете увидеть страницы текста! Если необходимо установить определенную версию Underscore, то номер версии можно задать явно.

```
$ npm install underscore@1.8.0
underscore@1.8.0 node_modules\underscore
```

Так где же этот модуль был фактически установлен? Если вы заглянете в свой каталог, то увидите новый подкаталог, `node_modules`; любые устанавливаемые локальные модули помещаются в этот каталог. Попробуйте удалить каталог `node_modules`; через мгновение мы воссоздадим его.

Устанавливая модули, вы захотите так или иначе следить за ними; модули, которые вы устанавливаете (и используете), называются *зависимостями* (dependencies) вашего проекта. По мере развития вашего проекта вам понадобится простой способ узнать, от каких пакетов зависит ваш проект, а npm предоставляет его в файле с именем `package.json`. Вы не должны создавать этот файл сами, достаточно выполнить команду **`npm init`** и ответить в интерактивном режиме на некоторые вопросы (вы просто будете нажимать на клавишу <Enter> в ответ на каждый вопрос и принимать стандартные значения; позже вы всегда сможете отредактировать файл и изменить свои ответы). Попробуйте это сейчас и обратите внимание на созданный файл `package.json`.

Зависимости разделяются на обычные зависимости и *зависимости времени разработки* (dev dependencies). Зависимости времени разработки — это пакеты, без которых ваше приложение может выполняться, но полезные или необходимые для разработки вашего проекта (мы скоро увидим их примеры). Таким образом, устанавливая локальные пакеты, следует добавить флаг `--save` или `--saveDev`; в противном случае пакет будет установлен, но не записан в файл `package.json`. Давайте снова установим Underscore с флагом `--save`.

```
$ npm install --save underscore
npm WARN package.json lj@1.0.0 No description
npm WARN package.json lj@1.0.0 No repository field.
underscore@1.8.3 node_modules\underscore
```

Вы могли бы задаться вопросом "Что это за предупреждения?" npm оповещает об отсутствии некоторых компонентов из вашего пакета. В этой книге вы можете игнорировать такие предупреждения: о них стоит волноваться, если вы используете npm для публикации собственных пакетов, а это выходит за рамки данной книги.

Если вы просмотрите свой файл `package.json` теперь, то увидите, что Underscore указан как зависимость. Идея управления зависимостями в том, что версии зависимостей, указанных в файле `package.json`, — это все, что необходимо для воссоздания (загрузки и установки) самой зависимости. Давайте опробуем это. Снова удалите каталог `node_modules`, а затем выполните команду `npm install` (обратите внимание, что мы не определяем конкретное имя пакета). npm установит все пакеты, указанные

в файле `package.json`. Можете заглянуть во вновь созданный каталог `node_modules`, чтобы убедиться в этом.

Инструменты сборки: Gulp и Grunt

Для большинства разработок вам, вероятно, понадобится *инструмент сборки* (build tool), автоматизирующий повторяющиеся задачи, выполняемые в процессе разработки. В настоящее время двумя самыми популярными инструментами сборки для JavaScript являются *Grunt* и *Gulp*. Оба они способны создавать системы. Grunt на несколько лет старше, Gulp, поэтому пользователей у него больше, но Gulp быстро его нагоняет. Поскольку Gulp, кажется, набирает все большую и большую популярность среди начинающих программистов JavaScript, в этой книге мы будем использовать именно его, хотя я не готов сказать, что Gulp превосходит Grunt (или наоборот).

Сначала установим Gulp глобально с помощью следующей команды.

```
$ npm install -g gulp
```

Если вы работаете под управлением Linux или OS X, то для использования параметра `-g` (global — глобально) команды npm вам понадобятся расширенные права: `sudo install -g gulp`. У вас будет запрошен пароль и вам будут предоставлены права администратора (только для этой команды). Если вы находитесь в системе, администрируемой кем-то другим, то вам, возможно, придется попросить, чтобы вас внесли в файл `sudoers`.

Для каждой системы, в которой вы работаете, необходимо один раз установить Gulp глобально, а затем для каждого проекта — локально. Для этого в корневом каталоге проекта выполните команду **`npm install --save-dev gulp`** (Gulp — это пример зависимости времени разработки: ваше приложение не будет нуждаться в нем при выполнении, но вы будете использовать его в процессе разработки). Теперь, когда Gulp установлен, мы создаем файл `gulpfile.js`.

```
const gulp = require('gulp');
// Зависимости Gulp будут здесь
gulp.task('default', function() {
    // Задачи Gulp будут здесь
});
```

Фактически мы не настраиваем Gulp, чтобы *сделать* что-нибудь, но мы можем убедиться, что Gulp способен теперь успешно выполняться.

```
$ gulp
[16:16:28] Using gulpfile /home/joe/work/lj/gulpfile.js
[16:16:28] Starting 'default'...
[16:16:28] Finished 'default' after 68 мс
```

Пользователи Windows могут получить сообщение об ошибке "The build tools for Visual Studio 2010 (Platform Toolset = v100) cannot be found" (Инструменты сборки для Visual Studio 2010 (Комплект инструментальных средств платформы = v100) не найдены). Многие пакеты npm зависят от инструментов сборки Visual Studio. Вы можете получить бесплатную версию Visual Studio со *страницы загрузки* `https://www.visualstudio.com/ru/downloads/`. Установив Visual Studio, найдите в программных файлах "Developer Command Prompt" (Командная оболочка разработчика). В этой оболочке перейдите в свой корневой каталог проекта и попытайтесь установить Gulp снова. Теперь должно получиться лучше. Вы не обязаны продолжать использовать командную оболочку из Visual Studio, но это самый простой способ установки модулей npm, имеющих зависимости от Visual Studio.

Структура проекта

Прежде чем приступать к использованию Gulp и Babel для преобразования нашего кода ES6 в код ES5, необходимо подумать, куда мы собираемся помещать код в пределах нашего проекта. В разработке JavaScript нет универсального стандарта для компоновки проектов: они слишком разнообразны для этого. Обычно исходный код помещают в каталог `src` или `js`. Мы собираемся поместить наш исходный код в каталог `es6`; это совершенно ясно указывает, что мы пишем код ES6.

Поскольку большинство проектов включают и серверный (Node), и клиентский (браузер) коды, мы также разделим эти две категории. Серверный код расположится непосредственно в каталоге `es6` в нашем корневом каталоге проекта, а код, предназначенный для браузера, — в каталоге `public/es6` (по определению любой код JavaScript, посылаемый браузеру, является открытым (public), и это вполне обычное соглашение).

В следующем разделе мы преобразуем свой код ES6 в ES5, поэтому нам понадобится место для размещения кода ES5 (не нужно смешивать его с кодом ES6). Обычно этот код помещают в каталог `dist` (distribution — для распространения).

В результате ваш корневой каталог проекта будет выглядеть примерно так.

```
.git              # Git
.gitignore

package.json      # npm
node_modules

es6               # исходный код для Node
dist

public/           # исходный код для браузера
    es6/
    dist/
```

Транскомпиляторы

На момент написания книги двумя наиболее популярными транскомпиляторами были *Babel* и *Traceur*. Я использовал оба, и они показали себя весьма удобными и работоспособными. В настоящее время я более склоняюсь к Babel, и в этой книге мы будем использовать этот транскомпилятор. Итак, приступим!

Изначально Babel задуман как транскомпилятор из ES5 в ES6, но впоследствии развился в универсальный транскомпилятор, способный осуществлять множество разных преобразований, включая ES6, React и даже ES7. Начиная с Babel версии 6 преобразования больше не включаются в Babel. Для выполнения нашего преобразования из ES5 в ES6 необходимо установить преобразование ES6 и настроить Babel так, чтобы использовать его. Мы делаем эти настройки локальными для нашего проекта, поскольку вполне вероятно, что мы будем использовать ES6 в одном проекте, React — в другом, а ES7 (или некий другой вариант) — в третьем. Сначала мы установим набор ES6 (он же ES2015).

```
$ npm install --save-dev babel-preset-es2015
```

Затем мы создадим в корневом каталоге нашего проекта файл `.babelrc` (предваряющая точка означает, что обычно файл должен быть скрытым). Содержимое этого файла таково.

```
{ "presets": ["es2015"] }
```

При наличии этого файла в проекте Babel сразу понимает, что вы используете ES6.

Запуск Babel с Gulp

Теперь мы можем использовать Gulp фактически: давайте преобразуем код ES6, который мы напишем, в переносимый код ES5. Мы преобразуем весь код в каталогах `es6` и `public/es6` в код ES5, располагаемый в каталогах `dist` и `public/dist`. Мы используем пакет `gulp-babel`, поэтому начнем с его установки командой `npm install --save-dev gulp-babel`. Затем отредактируем файл `gulpfile.js`.

```
const gulp = require('gulp');
const babel = require('gulp-babel');

gulp.task('default', function() {
    // исходный код для Node
    gulp.src("es6/**/*.js")
        .pipe(babel())
        .pipe(gulp.dest("dist"));
    // исходный код для браузера
    gulp.src("public/es6/**/*.js")
        .pipe(babel())
        .pipe(gulp.dest("public/dist"));
});
```

Gulp использует концепцию *конвейера* (pipeline). Мы начинаем с сообщения Gulp о расположении интересующих нас файлов: `src ("es6/**/*.js")`. Вы могли бы задаться вопросом о `**`; это шаблон для "любого каталога, включая подкаталоги". Таким образом, этот фильтр выберет все файлы `.js` в каталоге `es6` и любых его подкаталогах, независимо от глубины. Затем мы *транспортируем* (pipe) эти файлы исходного кода в Babel, который преобразует их из ES6 в ES5. На завершающем этапе откомпилированный код ES5 транспортируется по назначению, в каталог `dist`. Gulp сохранит имена и структуру каталогов ваших файлов исходного кода. Например, файл `es6/a.js` будет откомпилирован в `dist/a.js`, а `es6/a/b/c.js` — в `dist/a/b/c.js`. Мы повторим тот же процесс для файлов в каталоге `public/es6`.

Мы еще не изучили ES6, но давайте создадим файл примера кода ES6 и удостоверимся в работоспособности нашей конфигурации Gulp. Создайте файл `es6/test.js`, содержащий часть новых средств ES6 (не расстраивайтесь, если не понимаете содержимое этого файла; впоследствии вы все поймете!).

```
'use strict';
// средство es6: блок "левых" объявлений
const sentences = [
    { subject: 'JavaScript', verb: 'is', object: 'great' },
    { subject: 'Elephants', verb: 'are', object: 'large' },
];
// средство es6: деструктуризация объекта
function say({ subject, verb, object }) {
    // средство es6: строки шаблона
    console.log('${subject} ${verb} ${object}');
}
// средство es6: for..of
for(let s of sentences) {
    say(s);
}
```

Теперь создайте копию этого файла в каталоге `public/es6` (вы можете изменить содержимое массива `sentences`, если хотите удостовериться, что ваши файлы отличаются). Теперь введите **`gulp`** и просмотрите каталоги `dist` и `public/dist`. Вы увидите файл `test.js` в обоих каталогах. Давайте откроем этот файл и посмотрим, чем он отличается от своего эквивалента ES6.

Теперь давайте попробуем запустить код ES6 непосредственно.

```
$ node es6/test.js
/home/ethan/lje3/es6/test.js:8
function say({ subject, verb, object }) {
         ^
SyntaxError: Unexpected token {
    at exports.runInThisContext (vm.js:53:16)
    at Module._compile (module.js:374:25)
```

```
    at Object.Module._extensions..js (module.js:417:10)
    at Module.load (module.js:344:32)
    at Function.Module._load (module.js:301:12)
    at Function.Module.runMain (module.js:442:10)
    at startup (node.js:136:18)
    at node.js:966:3
```

Переданные Node сообщения об ошибках могут быть иными, поскольку Node находится в процессе реализации средств ES6 (если вы читаете эту книгу в достаточно далеком будущем, то все может сработать полностью!). Теперь давайте запустим эквивалент ES5.

```
$ node dist\test.js
JavaScript is great
Elephants are large
```

Мы успешно преобразовали код ES6 в переносимый код ES5, который должен выполняться везде! На последнем этапе добавьте каталоги `dist` и `public/dist` в свой файл `.gitignore`: мы хотим следить за исходным кодом ES6, но не за файлами ES5, которые создаются из него.

Анализ

Вы проходитесь *липким роликом* (lint roller) по костюму или платью, прежде чем пойти на вечеринку или интервью? Конечно, вы ведь хотите выглядеть лучше. Аналогично вы можете *почистить* (lint) свой код, чтобы сделать его (а следовательно, и себя) выглядящим лучшее. *Анализатор* (linter) критически оценивает ваш код и сообщает о сделанных вами наиболее распространенных ошибках. Я писал программное обеспечение в течение 25 лет, но хороший анализатор все еще находит ошибки в моем коде прежде меня. Для новичка это неоценимый инструмент, способный предохранить вас от множества неприятностей.

Есть несколько анализаторов JavaScript, но я предпочитаю ESLint от Николаса Закаса (Nicholas Zakas). Установим ESLint.

```
npm install -g eslint
```

Прежде чем начать использовать ESLint, необходимо создать файл конфигурации `.eslintrc` для нашего проекта. В каждом вашем проекте могут быть использованы разные технологии или стандарты, и файл `.eslintrc` позволяет ESLint анализировать ваш код соответственно.

Проще всего создать файл `.eslintrc`, выполнив команду `eslint --init`. В результате вам будет задано несколько вопросов в интерактивном режиме и стандартный файл будет создан автоматически.

В корневом каталоге проекта выполните команду `eslint --init`. Нужно будет дать ответы на следующие вопросы.

- Для выравнивания вы используете пробелы или табуляцию? Недавний опрос *StackOverflow* показал, что большинство программистов предпочитают табуляцию, но более опытные программисты предпочитают пробелы. Вы можете поступать по своему усмотрению.
- Для строк вы предпочитаете одиночные или двойные кавычки? Здесь ответ не имеет значения, мы хотим быть в состоянии использовать их одновременно.
- Какие окончания строк вы используете (Unix или Windows)? Если вы работаете под управлением Linux или OS X, выберите Unix, если под управлением Windows — то Windows.
- Точки с запятой нужны? Да.
- Вы используете средства ECMAScript 6 (ES6)? Да.
- Где выполняется ваш код (в Node или в браузере)? В идеале вы использовали бы разные конфигурации для кода браузера и Node, но это более сложная конфигурация. Просто выберите Node.
- Вы хотите использовать JSX? Нет. (JSX — это расширение JavaScript на базе XML, используемое в библиотеке UI `React` для Facebook. Мы не будем использовать его в этой книге.)
- Каким должен быть формат файла конфигурации (JSON или YAML)? Выберите JSON (YAML — популярный формат сериализации данных, как и JSON, но JSON лучше подходит для разработки JavaScript).

Ответив на все вопросы, вы получите файл `.eslintrc` и сможете начать использовать ESLint.

Есть несколько способов запустить ESLint. Вы можете запустить его непосредственно (например, `eslint es6/test.js`), интегрировать его в свой редактор или добавить в свой файл `gulpfile.js`. Интеграция редактора лучше, но инструкции зависят от конкретного редактора и конкретной операционной системы. Если вы хотите интеграцию в редактор, я рекомендую поискать в Google имя вашего редактора со словом "eslint".

Независимо от интеграции в редактор, я рекомендую добавить ESLint в ваш файл `gulpfile.js`. В конце концов, мы запускаем Gulp, когда готовы к сборке. Таким образом, это самое время проверить качество нашего кода. Сначала выполните команду

```
npm install --save.dev gulp-eslint
```

Затем измените файл `gulpfile.js`.

```
const gulp = require('gulp');
const babel = require('gulp-babel');
const eslint = require('gulp-eslint');

gulp.task('default', function() {
```

```
    // запуск ESLint
    gulp.src(["es6/**/*.js", "public/es6/**/*.js"])
        .pipe(eslint())
        .pipe(eslint.format());
    // исходный код для Node
    gulp.src("es6/**/*.js")
        .pipe(babel())
        .pipe(gulp.dest("dist"));
    // исходный код для браузера
    gulp.src("public/es6/**/*.js")
        .pipe(babel())
        .pipe(gulp.dest("public/dist"));
});
```

Теперь давайте посмотрим, что не нравится ESLint в нашем коде. Поскольку мы включали ESLint в нашу задачу `default`, мы можем просто запустить Gulp.

```
$ gulp
[15:04:16] Using gulpfile ~/git/gulpfile.js
[15:04:16] Starting 'default'...
[15:04:16] Finished 'default' after 84 ms
[15:04:16]
/home/ethan/lj/es6/test.js
    4:59  error Unexpected trailing comma comma-dangle
    9:5   error Unexpected console statement no-console

r 2 problems (2 errors, 0 warnings)
```

Понятно, что мы с Николасом Закасом не пришли к согласию по поводу завершающих запятых. К счастью, ESLint позволяет сделать собственный выбор о том, что является ошибкой, а что нет. Стандартным значением правила `comma-dangle` является `"never"`, но его можно отключить в целом или изменить на `"always-multiline"` (мой выбор). Отредактируйте файл `.eslintrc`, чтобы изменить этот параметр (но если вы согласны с Николасом по поводу завершающих запятых, то можете оставить стандартное `"never"`). Каждое правило в файле `.eslintrc` — это массив. Первый элемент — число, где 0 отключает правило, 1 требует выдать при невыполнении предупреждение, а 2 — считать это ошибкой.

```
{
    "rules": {
        /* Изменение стандартного правила comma-dangle... по иронии
           мы не можем использовать потерянные запятые здесь,
           поскольку это файл JSON. */
        "comma-dangle": [
            2,
            "always-multiline"
        ],
        "indent": [
```

```
        2,
        4
    ],
/* ... */
```

Если запустить `gulp` снова, то наша висящая запятая больше не будет вызывать ошибку. Фактически, если мы удалим ее, то получим ошибку!

Вторая ошибка относится к использованию файла `console.log`, который обычно считают "неустойчивым" (sloppy) (даже опасным, если вы пишете для устаревших браузеров), когда он используется в рабочем коде браузера. В учебных целях вы можете отключить его, поскольку мы будем использовать файл `console.log` повсюду в этой книге. Кроме того, вы, вероятно, захотите выключить правило `"quotes"`. Это я оставлю читателю как упражнение.

ESLint имеет много параметров настройки; они все полностью описаны на вебсайте ESLint.

Теперь, когда мы можем написать код ES6, транскомпилировать его в переносимый код ES5 и проанализировать его, чтобы улучшить, мы готовы погрузиться в изучение ES6!

Заключение

В этой главе мы узнали, что поддержка ES6 распространена еще не слишком широко, но это не мешает вам воспользоваться преимуществами ES6 сегодня, поскольку вы можете транскомпилировать свой код ES6 в переносимый код ES5.

При наладке новой машины для разработки вам понадобится следующее.

- Хороший текстовый редактор (см. главу 1).
- Git (инструкции по установке приведены на `https://git-scm.com/`).
- Gulp (`npm install -g gulp`).
- ESLint (`npm install -g eslint`).

Когда вы начнете новый проект (будь то проект для отработки примеров из этой книги или реальный проект), понадобятся следующие компоненты.

- Отдельный каталог для вашего проекта; мы называем его *корневым каталогом проекта*.
- Хранилище Git (`git init`).
- Файл `package.json` (`npm init`).
- Файл `gulpfile.js` (используйте созданный в этой главе).
- Локальные пакеты Gulp и Babel (`npm install --save-dev gulp gulp-babel babel-preset-es2015`).

- Файл `.babelrc` (содержимое: `{ "presets": ["es2015"] }`).
- Файл `.eslintrc` (используйте для его создания команды `eslint --init`, а затем отредактируйте согласно своим предпочтениям).
- Подкаталог для исходного кода Node (`es6`).
- Подкаталог для исходного кода браузера (`public/es6`).

Как только будет все установлено, ваш базовый рабочий цикл будет выглядеть следующим образом.

1. Вносите логически единообразные, связанные изменения.
2. Запустите Gulp, чтобы проверить и проанализировать код.
3. Повторяйте до тех пор, пока ваши изменения не будут работать без ошибок.
4. Проверьте и удостоверьтесь, что не собираетесь фиксировать ненужные изменения (`git status`). Если есть файлы ненужные в Git, добавьте их в свой файл `.gitignore`.
5. Внесите все свои изменения в Git (`git add -A`; если вы не хотите вносить все изменения, используйте вместо этого команду `git add` для каждого файла).
6. Фиксируйте свои изменения (`git commit -m "<`*описание внесенных изменений*`>"`).

В зависимости от проекта могут быть и другие этапы, такие как проверки (обычно как задача Gulp) и передача вашего кода в совместно используемое хранилище, такое как GitHub или Bitbucket (`git push`). Однако этапы, перечисленные здесь, встречаются в большинстве проектов.

В остальной части книги мы представляем исходный код, не повторяя этапы, необходимые для его сборки и запуска. Если код примера не предназначен явно для браузера, то все примеры кода должны выполняться в Node. Так, например, если дан пример `example.js`, вы помещаете этот файл в каталог `es6` и запускаете командой

```
$ gulp
$ node dist/example.js
```

Вы также можете пропустить этап Gulp и запустить его непосредственно с `babel-node` (хотя вы не сэкономите время, поскольку `babel-node` также требует транскомпиляции).

```
$ babel-node es6/example.js
```

Теперь пришло время изучать сам JavaScript!

Литералы, переменные, константы и типы данных

Эта глава — о *данных* и об их преобразовании в формат, понятный JavaScript.

Вы, вероятно, знаете, что все данные в конечном счете представляются на компьютере как длинные последовательности нулей и единиц, но для большинства повседневных задач нам нужно более естественное представление данных, в виде чисел, текста, дат и т.д. Мы называем эти абстракции *типами данных* (data type).

Прежде чем мы погрузимся в типы данных, доступные для JavaScript, давайте обсудим *переменные*, *константы* и *литералы* — доступные для нас в JavaScript механизмы *хранения* данных.

Обучаясь программированию, зачастую упускают важность словаря. Хотя, на первый взгляд, важность понимания того, чем литерал отличается от значения или оператор от выражения, может не казаться очевидной, незнание этих терминов воспрепятствует вашей способности к обучению. Большинство этих терминов не является специфическим для JavaScript, они общеприняты в информатике. Хорошее понимание этих концепций, конечно, важно, но, уделив внимание словарю, вы облегчите себе переход на другие языки.

Переменные и константы

Переменная (variable) — это, по существу, именованное значение, и, как подразумевает название, данное значение может изменяться в любое время. Например, если мы работаем над системой контроля климата, у нас могла бы быть переменная `currentTempC`.

```
let currentTempC = 22; // градусы Цельсия
```

Ключевое слово `let` — это новшество ES6; до ES6 единственной возможностью было ключевое слово `var`, которое мы обсудим в главе 7.

Этот оператор делает две вещи: объявляет (создает) переменную `currentTempC` и присваивает ей исходное значение. Мы можем изменить значение переменной `currentTempC` в любой момент.

```
currentTempC = 22.5;
```

Обратите внимание: мы не используем `let` снова; ключевое слово `let` зарезервировано для объявления переменной, и вы можете сделать это только однажды.

Для чисел нет никакого способа ассоциировать единицы со значением. Таким образом, у языка JavaScript нет никакого способа указать нам, что в переменной `currentTempC` хранится значение в градусах Цельсия, и если мы присвоим ей значение в градусах Фаренгейта, то произойдет ошибка. Поэтому, чтобы прояснить, что единицами являются градусы Цельсия, я решил добавить суффикс "`C`" к имени переменной. Хотя синтаксис самого языка JavaScript не обязывает вас это делать, подобная форма документирования предотвращает случайные ошибки.

Объявляя переменную, вы не обязаны назначать ей исходное значение. Если вы не сделаете этого, то она неявно получит специальное значение, `undefined`.

```
let targetTempC; // эквивалент "let targetTempC = undefined";
```

Вы можете также объявить несколько переменных в том же операторе `let`.

```
let targetTempC, room1 = "conference_room_a", room2 = "lobby";
```

В этом примере мы объявили три переменные: `targetTempC`, без начального значения, а следовательно, с неявным присваиванием значения `undefined`; `room1`, с исходным значением `"conference_room_a"`; и `room2`, с исходным значением `"lobby"`. Переменные `room1` и `room2` — это примеры *строковых* (или текстовых) переменных.

Константа (нововведение ES6) также содержит значение, но в отличие от переменной оно не может быть изменено после инициализации. Давайте используем константы для выражения комфортной комнатной температуры и максимальной температуры (ключевое слово `const` также позволяет объявить несколько констант).

```
const ROOM_TEMP_C = 21.5, MAX_TEMP_C = 30;
```

Написание имен констант прописными буквами и символами подчеркивания весьма распространено, но не обязательно. Так их проще заметить в коде и сразу понять, что вы не должны пытаться изменять их значение.

Переменные или константы: что использовать?

Вообще, константы предпочтительнее переменных. Как правило, вам нужно удобное имя для некой части данных, нзначение которой не изменяется. Преимущество

использования констант в том, что они предотвращают случайное изменение значения того, что не должно измениться. Например, если вы работаете над частью своей программы, которая выполняет некое действие с пользователем, у вас может быть переменная по имени `user`. Если вы имеете дело только с одним пользователем, то изменение значения `user` привело бы к ошибке в коде. Если вы работаете с двумя пользователями, то вы могли бы назвать их `user1` и `user2` вместо многократного использования одной переменной `user`.

Таким образом, эмпирическое правило гласит, что предпочтительней использовать константу; но если появится насущная потребность изменить значение константы, то вы всегда можете заменить ее переменной.

Есть одна ситуация, в которой нужно использовать переменные, а не константы: в элементе управления циклом (о котором мы узнаем в главе 4). Без переменных не обойтись и в других ситуациях, когда некое значение естественным образом изменяется со временем (такие, как `targetTempC` и `currentTemp` в этой главе). Если вы возьмете в привычку использовать константы, то скоро удивитесь тому, как редко вам могут понадобиться переменные.

В примерах этой книги я попытался использовать константы вместо переменных везде, где было возможно.

Именование идентификаторов

Имена переменных и констант (а также имена функций, которые мы будем рассматривать в главе 6) являются *идентификаторами* и у них есть правила именования.

- Идентификаторы должны начинаться с символа, знака доллара (`$`) или символа подчеркивания (`_`).
- Идентификаторы состоят из символов, чисел, знаков доллара (`$`) и символов подчеркивания (`_`).
- Символы Unicode допустимы (например, `π` или `ö`).
- Зарезервированные слова (см. приложение A) не могут быть идентификаторами.

Обратите внимание, что знак доллара — это не специальный символ, как в некоторых других языках: это просто один из символов, который вы можете использовать в именах идентификаторов (многие библиотеки, такие как jQuery, используют знак доллара как отдельный идентификатор).

Зарезервированные слова являются частью языка JavaScript, поэтому у вас не может быть переменной `let`, например.

Нет единого соглашения для идентификаторов JavaScript, но наиболее распространены два.

Верблюжья нотация (camel case), например `currentTempC`, `anIdentifierName` (называется так потому, что заглавные буквы выглядят, как горбы на спине верблюда).

Змеиная нотация (snake case), например `current_temp_c`, `an_identifier_name` (немного менее популярна).

Вы можете использовать любое соглашение по своему предпочтению, но единообразие является хорошей идеей: выберите одно и придерживайтесь его. Если вы работаете в группе или делаете свой проект доступным сообществу, постарайтесь придерживаться уже принятого соглашения.

Также имеет смысл придерживаться следующих соглашений.

- Идентификаторы не должны начинаться с заглавной буквы *за исключением* имен классов (мы будем рассматривать их в главе 9).
- Как правило, идентификаторы, начинающиеся с одного или двух символов подчеркивания, используются для специальных или "внутренних" переменных. Если необходимо создать собственную специальную категорию переменных, не начинайте их имена с символа подчеркивания.
- При использовании jQuery начинающиеся со знака доллара идентификаторы традиционно относятся к объектам в оболочке jQuery (см. главу 19).

Литералы

Мы уже видели некоторые *литералы* (literal): когда мы присваивали значение переменной `currentTempC`, мы предоставили *числовой литерал* (`22` при инициализации и `22.5` в следующем примере). Аналогично, когда мы инициализировали переменную `room1`, мы предоставляли *строковый литерал* (`"conference_room_a"`). Слово *литерал* означает, что вы предоставляете значение непосредственно в программе. По существу, литерал — это средство *создания значения*; JavaScript получает предоставленное вами литеральное значение и создает из него значение данных.

Важно понимать различие между *литералом* и *идентификатором*. Вспомните, например, прежний пример, в котором мы создали переменную `room1`, у которой было значение `"conference_room_a"`. Здесь `room1` — это идентификатор (именующий переменную), а `"conference_room_a"` — это строковый литерал (а также значение переменной `room1`). JavaScript в состоянии отличить идентификатор от литерала по наличию кавычек (числа не нуждаются в кавычках, поскольку идентификаторы не могут начинаться с цифр). Рассмотрим следующий пример.

```
let room1 = "conference_room_a"; // "conference_room_a" (в кавычках)
                                 // - это литерал

let currentRoom = room1;         // currentRoom имеет теперь
                                 // то же значение, что
                                 // и room1 ("conference_room_a")
```

```
let currentRoom = conference_room_a; // приводит к ошибке; никакого
                                     // идентификатора conference_room_a
                                     // не существует
```

Вы можете использовать литерал везде, где можно использовать идентификатор (где ожидается значение). Например, в нашей программе мы могли бы использовать числовые литералы `21.5` повсюду вместо `ROOM_TEMP_C`. Если вы используете числовой литерал в паре мест, то все в порядке. Но если вы используете его в 10 или 100 местах, то следует применить вместо него константу или переменную: это облегчит чтение вашего кода и позволит изменять значения в одном месте вместо многих.

Это вам как программисту решать, что сделать переменной, а что константой. Некоторые вещи — вполне очевидные константы (такое, например, как приблизительное значение числа π (отношение периметра окружности к его диаметру) или `DAYS_IN_MARCH`). Другие вещи, такие как `ROOM_TEMP_C`, не совсем очевидны: 21,5°C может быть вполне комфортной комнатной температурой для меня, но не обязательно для вас. Поэтому если это значение перестраиваемо в вашем приложении, то вы сделали бы его переменной.

Базовые типы и объекты

В JavaScript значения бывают или *базовыми типами* (primitive), или *объектами* (object). Базовые типы (такие, как строки и числа) *неизменны* (immutable). Число 5 всегда будет числом 5; строка `"alpha"` всегда будет строкой `"alpha"`. Для чисел это кажется очевидным, но не для строк: когда строки связывают вместе (`"alpha" + "omega"`), они иногда выглядят, как та же строка, только измененная. Но это не так: получается новая строка, отличная от прежней, как число 6 отличается от числа 5. Существует шесть базовых типов, которые мы и рассмотрим.

- Число (Number).
- Строка (String).
- Логический (Boolean).
- Пусто (Null).
- Неопределенный (Undefined).
- Символ (Symbol).

Обратите внимание, что неизменность не означает, что *содержимое переменной* не может быть изменено.

```
let str = "hello";
str = "world";
```

Сначала переменная `str` инициализируется (неизменяемым) значением `"hello"`, а затем ей присваивается новое (неизменяемое) значение `"world"`. Важнее всего то, что строки `"hello"` и `"world"` — разные; изменилось только значение, хранимое в переменной `str`. Обычно это различие является чисто академическим, но знание данного факта пригодится позже, при обсуждении функций в главе 6.

Кроме этих шести базовых типов, существуют *объекты* (object). В отличие от базовых типов, объекты способны принимать различные формы и значения, они подобны хамелеону.

Благодаря гибкости объекты применяются для создания специальных типов данных. Фактически JavaScript предоставляет несколько объектов встроенных типов. Мы рассмотрим следующие объекты встроенных типов.

- `Array`.
- `Date`.
- `RegExp`.
- `Map` и `WeakMap`.
- `Set` и `WeakSet`.

И наконец у базовых типов чисел, строк и логических значений есть соответствующие типы среди объектов: `Number`, `String` и `Boolean`. Эти соответствующие объекты не хранят значения фактически (как это делают базовые типы), а скорее предоставляют функциональные возможности, связанные с соответствующим базовым типом. Мы обсудим объекты этих типов наряду с соответствующими им базовыми.

Числа

Некоторые числа, такие как 3, 5,5 и 1 000 000, могут быть представлены компьютером точно, но многие числа требуют приближения. Например, число π не может быть представлено компьютером вообще, поскольку количество его цифр бесконечно и они не повторяются. Другие числа, такие как 1/3, могут быть представлены специальными методиками, но из-за бесконечного повторения десятичных чисел (3,33333...) они обычно также округляются.

В JavaScript, наряду с большинством других языков программирования, используется приближенное представление вещественных чисел в формате *IEEE 764* (*числа с плавающей точкой двойной точности*), который далее я буду называть просто *двойной точностью*. Детали этого формата не рассматриваются в данной книге, да вам особо это и не нужно, если, конечно, вы не занимаетесь сложным анализом чисел. Однако последствия округлений, неизбежных в этом формате, зачастую сказываются. Например, если вы вычислите в JavaScript выражение `0.1 + 0.2`, то получите `0.30000000000000004`. Это не значит, что JavaScript "сломался" или плох

в математике: это просто неизбежное последствие приближения бесконечных значений в конечной области памяти.

JavaScript — необычный язык программирования, в нем есть только этот один числовой тип данных[1]. В большинстве языков есть несколько целочисленных типов и два или более типов с плавающей точкой. С одной стороны, это упрощает JavaScript, особенно для новичков. С другой стороны, это снижает пригодность JavaScript для определенных приложений, требующих эффективной целочисленной арифметики или чисел с фиксированной точностью.

JavaScript распознает четыре типа числовых литералов: десятичный, двоичный, восьмеричный и шестнадцатеричный. Десятичные литералы позволяют выразить целые числа (без десятичной части), десятичные числа и числа в экспоненциальной форме записи по основанию 10 (научная форма записи). Кроме того, есть специальные значения для бесконечности, отрицательной бесконечности и "не числа" (технически это не числовые литералы, но в действительности они приводят к числовым значениям, поэтому я включаю их сюда).

```
let count = 10;          // целочисленный литерал; count все еще
                         // двойной точности
const blue = 0x0000ff;   // шестнадцатеричное (hex ff = decimal 255)
const umask = 0o0022;    // восьмеричное (octal 22 = decimal 18)
const roomTemp = 21.5;   // десятичное число
const c = 3.0e6;         // экспоненциальное (3.0 × 10^6 = 3,000,000)
const e = -1.6e-19;      // экспоненциальное
                         // (-1.6× 10^-19 = 0.00000000000000000016)
const inf = Infinity;
const ninf = -Infinity;
const nan = NaN;         // "не число"
```

Независимо от формата используемого литерала (десятичный, шестнадцатеричный, экспоненциальный и т.д.), создаваемое число сохраняется в том же формате: двойной точности (double). Различные литеральные форматы просто позволяют определять числа в любом удобном формате. В JavaScript ограничена поддержка отображения чисел в различных форматах, как мы обсудим в главе 16.

Математики могли бы возразить: бесконечность — не число! Действительно, они правы, но, разумеется, бесконечность — это и не `NaN`. Данные понятия не относятся к числам, с которыми можно выполнять вычисления; скорее они являются некими индикаторами.

Кроме того, у соответствующего объекта `Number` есть несколько полезных свойств, способных представлять важные числовые значения.

[1] В будущем все может измениться: специальные целочисленные типы — это наиболее широко обсуждаемое средство языка.

```
const small = Number.EPSILON;  // наименьшее значение, которое может быть
                               // добавлено к 1, чтобы получить отличное
                               // от нее число, большее, чем 1.
                               // Приблизительно 2.2e-16
const bigInt = Number.MAX_SAFE_INTEGER; // наибольшее допустимое целое
                                        // число
const max = Number.MAX_VALUE;           // наибольшее допустимое число
const minInt = Number.MIN_SAFE_INTEGER; // наименьшее допустимое целое
                                        // число
const min = Number.MIN_VALUE;           // наименьшее допустимое число
const nInf = Number.NEGATIVE_INFINITY;  // то же, что и -Infinity
const nan = Number.NaN;                 // то же, что и NaN
const inf = Number.POSITIVE_INFINITY;   // то же, что и Infinity
```

Мы обсудим важность этих значений в главе 16.

Строки

Строка (string) — это просто текстовые данные (слово *строка* происходит от "строки символов" — термина, впервые использованного в конце 1800-х годов типографскими наборщиками, а затем математиками для представления последовательности символов в определенном порядке).

Строки в JavaScript представлены в формате *Unicode*. Это индустриальный компьютерный стандарт для представления текстовых данных, включающий *точки кода* (code point) для каждой буквы или символа в наиболее известных человеческих языках (включая "языки", которые могли бы удивить вас, такие как Emoji). Хотя Unicode способен представлять текст на любом языке, это не означает, что программное обеспечение, визуализирующее символы Unicode, будет способно правильно визуализировать каждую кодовую точку. В этой книге мы будем придерживаться довольно общих символов Unicode, которые, вероятней всего, доступны в вашем браузере и на консоли. Если вы будете работать с экзотическими символами или языками, то проведите дополнительное исследование и выясните, визуализируются ли необходимые кодовые точки Unicode.

В JavaScript строковые литералы представляются в одинарных кавычках, парных кавычках или обратных апострофах. *Обратные апострофы* (backtick) были введены в ES6 для поддержки *строковых шаблонов* (template string), которые мы рассмотрим ниже.

Экранирование специальных символов

Когда вы пытаетесь представить текстовые данные в программе, также состоящей из текстовых данных, возникает проблема — как отличить текстовые данные от самой программы? Заключение строк в кавычки является началом, но что если необходимо использовать кавычки *в* строке? Для решения этой проблемы применяется *экранирование* (escaping), указывающее, что следующий символ не завершает строку. Рассмотрим следующие примеры (которые не требуют экранирования).

```
const dialog = 'Sam looked up, and said "hello, old friend!", as Max walked in.';
const imperative = "Don't do that!";
```

В строке `dialog` мы можем использовать двойные кавычки без страха, поскольку наша строка заключена в одинарные кавычки. Аналогично в строке `imperative` мы можем использовать апостроф, поскольку строка заключена в двойные кавычки. Но что если бы понадобилось использовать оба? Посмотрите.

```
// это приведет к ошибке
const dialog = "Sam looked up and said "don't do that!" to Max.";
```

Эта строка `dialog` приведет к ошибке независимо от того, какие кавычки мы выбираем. К счастью, мы можем экранировать кавычки, используя символ обратной косой черты (\), который является сигналом для JavaScript, что строка *не* заканчивается. Вот прежний пример, переписанный так, чтобы использовать оба типа кавычек.

```
const dialog1 = "He looked up and said \"don't do that!\" to Max.";
const dialog2 = 'He looked up and said "don\'t do that!" to Max.';
```

Затем, конечно, мы столкнемся с проблемой курицы и яйца, когда в строке мы захотим использовать обратную косую черту. Чтобы решить эту проблему, обратная косая черта может экранировать сама себя.

```
const s = "In JavaScript, use \\ as an escape character in strings.";
```

Какие кавычки использовать, одинарные или двойные, — это ваше дело. Я обычно предпочитаю парные кавычки, когда пишу текст, представляемый пользователю, поскольку я использую сокращения (например, *don't*) чаще, чем парные кавычки. Когда я пишу код HTML в строке JavaScript, я предпочитаю одинарные кавычки, чтобы использовать парные кавычки для значений атрибутов.

Специальные символы

Обратная косая черта используется не только для экранирования кавычек, но и для представления неотображаемых символов, таких как перевод строки и произвольные символы Unicode. Общепринятые специальные символы приведены в табл. 3.1.

Таблица 3.1. Общепринятые специальные символы

Код	Описание	Пример
`\n`	Новая строка (технически знак перевода строки: ASCII/Unicode 10)	`"Line1\nLine2"`
`\r`	Символ возврата каретки (ASCII/Unicode 13)	`"Windows line 1\r\nWindows line 2"`
`\t`	Табуляция (ASCII/Unicode 9)	`"Speed:\t60kph"`
`\'`	Одинарная кавычка (обратите внимание, что вы можете использовать данную форму записи даже тем, где это совсем не требуется)	`"Don\'t"`

Код	Описание	Пример
`\"`	Двойная кавычка (обратите внимание, что вы можете использовать данную форму записи даже тем, где это совсем не требуется)	`'Sam said \"hello\".'`
`\'`	Обратный апостроф (или "ударение"; нововведение ES6)	`'New in ES6: \' strings.'`
`\$`	Знак доллара (нововведение ES6)	`'New in ES6: ${interpolation}'`
`\\`	Обратная косая черта	`"Use \\\\ to represent \\!"`
`\u`*XXXX*	Произвольная кодовая точка Unicode (где *XXXX* — это шестнадцатеричный код точки)	`"De Morgan's law: \u2310(P \ u22c0 Q) \u21D4 (\u2310P) \ u22c1 (\u2310Q)"`
`\x`*XX*	Символ "Latin-1" (где *XX* — это шестнадцатеричный код точки Latin-1)	`"\xc9p\xe9e is fun, but foil is more fun."`

Обратите внимание, что набор символов Latin-1 — это подмножество Unicode, и любой его символ `\x`*XX* может быть представлен эквивалентной кодовой точкой Unicode `\u00`*XX*. Для шестнадцатеричных чисел вы можете использовать строчные или прописные буквы как вам нравится; лично я предпочитаю нижний регистр, поскольку мне так легче читать.

Вы не обязаны использовать управляющие коды для символов Unicode; можете просто ввести их непосредственно в своем редакторе. Способ доступа к символам Unicode зависит от конкретного редактора и операционной системы (как правило есть несколько способов); если желательно вводить символы Unicode непосредственно, обратитесь к документации на ваш редактор или операционную систему.

Кроме того, есть несколько редко используемых специальных символов, представленных в табл. 3.2. Я не помню, чтобы использовал их когда-либо в программе JavaScript, но я привожу их здесь для порядка.

Таблица 3.2. Редко используемые специальные символы

Код	Описание	Пример
`\0`	**Пустой символ, NUL (ASCII/Unicode 0)**	`"ASCII NUL: \0"`
`\v`	Вертикальная табуляция (ASCII/Unicode 11)	`"Vertical tab: \v"`
`\b`	Забой (возврат на один символ, ASCII/Unicode 8)	`"Backspace: \b"`
`\f`	Прогон страницы (ASCII/Unicode 12)	`"Form feed: \f"`

Строковые шаблоны

Очень часто в строках приходится выражать значения. Для этого можно применить механизм *конкатенации строк* (string concatenation).

```
let currentTemp = 19.5;
// 00b0 - код Unicode для символа "градуса"
const message = "The current temperature is " + currentTemp + "\u00b0C";
```

Вплоть до ES6 конкатенация строк была для этого единственным способом (за исключением применения библиотек стороннего производителя). ES6 вводит *строковые шаблоны* (template string), или *строковую интерполяцию* (string interpolation). Строковые шаблоны позволяют быстро вводить значения в строку. В строковых шаблонах используются обратные апострофы вместо одинарных или двойных кавычек. Вот предыдущий пример, переписанный с использованием строковых шаблонов.

```
let currentTemp = 19.5;
const message = 'The current temperature is ${currentTemp}\u00b0C';
```

В строковом шаблоне знак доллара становится специальным символом (поэтому в случае использовании его в виде литерала, вы должны экранировать его с помощью обратной косой черты): если он сопровождается значением[2], заключенным в фигурные скобки, то значение вставляется в строку.

Строковые шаблоны — это одно из моих любимых средств ES6, и вы увидите их повсюду в этой книге.

Поддержка многострочных строк

До ES6 поддержка многострочных (multiline) строк была, в лучшем случае, слабой. Спецификация языка позволяла использовать экранированный символ новой строки в конце строки исходного кода, но я никогда не использовал это средство в связи с ненадежной поддержкой браузером. После принятия стандарта ES6 это средство, вероятнее всего, будет доступно, но есть некоторые странности, о которых вы должны знать. Обратите внимание, что эти методики, вероятно, не будут работать на консоли JavaScript (как и в вашем браузере), поэтому фактически вы должны будете написать файл JavaScript, чтобы испытать их. В строки в одиночных и парных кавычках вы можете вставить символ новой строки таким образом.

```
const multiline = "line1\
line2";
```

Если вы ожидаете, что `multiline` будет строкой с символом новой строки внутри, то вы будете удивлены: наклонная черта в конце строки экранирует символ новой строки, но не вставляет его в строку. В результате получится `"line1line2"`. Если вам нужна фактически новая строка, то следует сделать так.

```
const multiline = "line1\n\
line2";
```

[2] Фактически в фигурных скобках можно использовать любое *выражение*. Выражения мы рассмотрим в главе 5.

Строки с обратным апострофом ведут себя немного естественней.

```
const multiline = 'line1
line2';
```

В результате получится строка с новой строкой. Однако в обоих случаях любой отступ в начале строки будет включен в результирующую строку. В следующем примере создана строка с новыми строками *и* отступом перед `line2` и `line3`, что может быть нежелательно.

```
const multiline = 'line1
    line2
    line3';
```

Поэтому я избегаю многострочного синтаксиса: это вынуждает меня или отказаться от отступа, повышающего читабельность кода, или включать отступ в многострочные строки, что не всегда хорошо. Если я действительно хочу разбить строку на несколько строк, я обычно использую конкатенацию.

```
const multiline = "line1\n" +
    "line2\n" +
    "line3";
```

Это позволяет мне выравнивать свой код для облегчения чтения и получить желаемую строку. Обратите внимание: вы можете смешивать типы строк при конкатенации.

```
const multiline = 'Current temperature:\n' +
    '\t${currentTemp}\u00b0C\n' +
    "Don't worry...the heat is on!";
```

Числа как строки

Если вы помещаете число в кавычки, то это не число, а строка. Как уже упоминалось, при необходимости JavaScript автоматически преобразует строки, содержащие числа в числа. Как и когда это происходит, не всегда очевидно; мы обсудим данную тему в главе 5. В следующем примере показано, когда это преобразование происходит, а когда нет.

```
const result1 = 3 + '30'; // 3 преобразуется в строку;
                          // результат - строка '330'
const result2 = 3 * '30'; // '30' преобразуется в число;
                          // результат - число 90
```

Эмпирическое правило: когда вы хотите использовать числа, используйте числа (т.е. не ставьте кавычки), а когда хотите использовать строки, используйте строки. В неопределенной ситуации, например, когда вы обрабатываете введенные пользователем данные, которые всегда поступают в виде строки, вам следует преобразовать их в число, где это нужно. Далее в этой главе мы обсудим методики преобразования типов данных.

Логические значения

Переменная логического типа имеет только два возможных значения: `true` и `false`. Некоторые языки (например, C) используют вместо логических значений числа: 0 — это `false`, а любое другое число — `true`. В JavaScript есть подобный механизм, позволяющий рассматривать *любое* значение (не только числа) как "истину" или "ложь", что мы обсудим далее, в главе 5.

Остерегайтесь использовать кавычки, когда намереваетесь указать логическое значение. В частности, многих удивляет тот факт, что строка `"false"` фактически является истиной! Вот правильный способ выражения логических литералов.

```
let heating = true;
let cooling = false;
```

Символы

Символы (symbol) являются новым типом данных ES6, представляющим индивидуальные лексемы. Вновь созданный символ уникален: ему не будет соответствовать никакой другой символ. Таким образом, символы походят на объекты (каждый объект уникален). Однако во всем остальном символы напоминают базовые типы. Это весьма полезное средство языка, допускающее расширение, о котором мы узнаем больше в главе 9.

Символы создаются конструктором[3] `Symbol()`. Вы можете, но не обязаны, снабдить его произвольным описанием, предназначенным только для удобства.

```
const RED = Symbol();
const ORANGE = Symbol("The color of a sunset!");
RED === ORANGE                  // ложь: каждый символ уникален
```

Я рекомендую использовать символы всякий раз, когда вам необходим уникальный идентификатор, который не должен быть по неосторожности перепутан с неким другим идентификатором.

Типы `null` и `undefined`

В JavaScript есть два специальных типа, `null` и `undefined`. У типа `null` есть только одно возможное значение (`null` — пусто), а у `undefined` — только `undefined` (неопределенно). Типы `null` и `undefined` представляют нечто, чего не существует, и даже факт наличия двух отдельных типов данных не прекратил недопонимания, особенно среди новичков.

[3] Если вы уже знакомы с объектно-ориентированным программированием в JavaScript, то обратите внимание, что создание символа с ключевым словом `new` не допускается и является исключением из соглашения, согласно которому идентификаторы, начинающиеся с заглавных букв, должны использоваться с оператором `new`.

Считается, что тип данных `null` предназначен для программистов, а тип `undefined` зарезервирован для самого JavaScript. Они означают, что некое значение еще отсутствует. Но это не обязательное правило: значение `undefined` вполне доступно для программистов, но здравый смысл требует использовать его чрезвычайно осторожно. Я явно присвоил переменной значение `undefined` только однажды, когда хотел преднамеренно подражать поведению переменной, которой еще не было присвоено значение. Однако обычно, когда вы хотите указать, что значение переменной не известно или не применимо, лучший выбор — значение `null`. Это может показаться придирками к мелочам, но иногда, особенно начинающему программисту, советуют использовать `null`, когда он не уверен. Обратите внимание, что, если вы объявляете переменную, не присваивая ей значение явно, стандартно у нее будет значение `undefined`. Вот примеры использования литералов `undefined` и `null`.

```
let currentTemp;          // неявное значение undefined
const targetTemp = null; // targetTemp null -- "еще не известно"
currentTemp = 19.5;       // currentTemp имеет теперь значение
currentTemp = undefined; // currentTemp выглядит, как будто она никогда
                          // не была инициализирована; не рекомендуется
```

Объекты

В отличие от неизменяемых базовых типов, способных содержать только одно представленное значение, объекты могут представлять несколько значений или сложные значения, а также изменять их в процессе существования. По сути, объект — это *контейнер*, и его содержимое может измениться со временем (это будет тот же объект с другим содержимым). Как и базовые типы, объекты имеют литеральный синтаксис: фигурные скобки (`{` и `}`). Поскольку фигурные скобки используются парой, это позволяет нам выразить содержимое объекта. Начнем с пустого объекта.

```
const obj = {};
```

Мы можем назвать свой объект как угодно по своему усмотрению, и обычно используется описательное имя, такое как `user` или `shoppingCart`. Мы только изучаем механику объектов, и наш пример не представляет ничего конкретного, поэтому мы и выбрали обобщенное имя `obj`.

Содержимое объекта называется *свойствами* (properties) или *членами* (members), а свойства состоят из *имени* (или *ключа*) и *значения*. Имена свойств должны быть строками или символами, а значения могут иметь любой тип (включая другие объекты). Давайте добавим в объект `obj` свойство `color`.

```
obj.size;                 // undefined
obj.color = "yellow" ;    // "yellow"
```

Чтобы использовать оператор обращения к члену класса, имя свойства должно быть корректным идентификатором. Если нужны имена свойств, не являющиеся допустимыми идентификаторами, используйте оператор *вычисляемого доступа к члену* (computed member access), применимый также и к допустимым идентификаторам.

```
obj["not an identifier"] = 3;
obj["not an identifier"];       // 3
obj["color"];                   // "yellow"
```

Оператор вычисляемого доступа к члену применим и для свойств символов.

```
const SIZE = Symbol();
obj[SIZE] = 8;
obj[SIZE];                      // 8
```

Здесь объект `obj` содержит три свойства с ключами: `"color"` (строка, являющаяся допустимым идентификатором), `"not an identifier"` (строка, не являющаяся допустимым идентификатором) и `SIZE` (символ).

Если вы используете консоль JavaScript, то можете заметить, что она не выводит символ `SIZE` как свойство объекта `obj`. Вы можете проверить это, введя **`obj[SIZE]`**. Дело в том, что символьные свойства рассматриваются иначе и не отображаются стандартно. Обратите также внимание на то, что ключ для этого свойства — символ `SIZE`, а *не* строка `"SIZE"`. Вы можете проверить это, введя **`obj.SIZE = 0`** (операторы доступа к члену *всегда* работают правильно для строковых свойств), а затем **`obj[SIZE]`** и **`obj.SIZE`** (или **`obj["SIZE"]`**).

Давайте остановимся и вспомним о различиях между базовыми типами и объектами. В этом разделе мы манипулируем объектом, содержащим переменную `obj`, и изменяем его, однако `obj` *все время был указателем на тот же объект.* Если бы `obj` вместо этого содержал строку, число или значение любого другого базового типа, то при его изменении это было бы каждый раз *другое* значение базового типа. Иными словами, `obj` указывает на тот же объект все время, но сам объект изменяется.

В экземпляре объекта `obj` мы создали пустой объект, но литеральный синтаксис объектов позволяет нам также создать объект, обладающий свойствами с самого начала. Свойства в фигурных скобках разделяются запятыми, а имена и значения — двоеточиями.

```
const sam1 = {
    name: 'Sam',
    age: 4,
};

const sam2 = { name: 'Sam', age: 4 }; // объявление в одной строке
```

```
const sam3 = {
    name: 'Sam',
    classification: {                  // значения свойств сами
        kingdom: 'Anamalia',           // могут быть объектами
        phylum: 'Chordata',
        class: 'Mamalia',
        order: 'Carnivoria',
        family: 'Felidae',
        subfaimily: 'Felinae',
        genus: 'Felis',
        species: 'catus',
    },
};
```

В этом примере мы создали три новых объекта, продемонстрировав литеральный синтаксис объектов. Обратите внимание, что объекты `sam1` и `sam2` содержат те же свойства, но это *два разных объекта* (еще одно отличие от базовых типов: две переменные, содержащие значение 3, относятся к одному и тому же типу). Свойство `classification` в объекте `sam3` само является объектом. Давайте рассмотрим различные способы доступа к семейству кота Сэма (здесь также не имеет значения, используем ли мы одиночные, двойные кавычки или обратные апострофы).

```
sam3.classification.family;        // "Felinae"
sam3["classification"].family;     // "Felinae"
sam3.classification["family"];     // "Felinae"
sam3["classification"]["family"]; // "Felinae"
```

Объекты могут также содержать *функции* (function). Подробно функции мы рассмотрим в главе 6, а пока достаточно знать, что функция содержит код (по существу, это подпрограмма). Вот как мы добавляем функцию в `sam3`.

```
sam3.speak = function() { return "Meow!"; };
```

Теперь мы можем *вызвать* эту функцию, добавив к ней круглые скобки.

```
sam3.speak();                      // «Meow!»
```

И наконец оператором `delete` мы можем удалить свойство из объекта.

```
delete sam3.classification; // все дерево classification удалено
delete sam3.speak;          // функция speak удалена
```

Если вы знакомы с *объектно-ориентированным программированием* (ООП), то можете задаться вопросом "Как объекты JavaScript соотносятся с ООП?" Пока вы можете считать объект контейнером; ООП мы обсудим в главе 9.

Объекты Number, String и Boolean

Как уже упоминалось, числа, строки и логические значения имеют соответствующие типы объектов (`Number`, `String` и `Boolean`). Эти объекты служат двум целям:

хранить специальные значения (такие, как `Number.INFINITY`) и обеспечивать функциональные возможности в форме функции. Рассмотрим следующий код.

```
const s = "hello";
s.toUpperCase();     // "HELLO"
```

В этом примере создается впечатление, что `s` — это объект (мы обратились к функциональному свойству, как будто оно есть). Но нам известно, что `s` — это базовый строковый тип. Так как же это произошло? Для этого JavaScript создает *временный* объект `String` (у которого есть функция `toUpperCase`, кроме других прочих). По завершении вызова функции JavaScript удаляет объект. Для доказательства давайте попробуем присвоить свойство строке.

```
const s = "hello";
s.rating = 3;        // ошибки нет... удача?
s.rating;            // undefined
```

JavaScript позволяет сделать это, создавая впечатление, что мы присваиваем свойство строке `s`. На самом деле свойство присваивается временному объекту `String`, который для этого и создается. Сразу после этого временный объект удаляется, поэтому результатом `s.rating` будет `undefined`.

Хотя такое поведение JavaScript применяется редко (если вообще применяется), о нем следует знать.

Массивы

Массивы (array) в JavaScript — это объекты специального типа. В отличие от обычных объектов содержимое массива упорядочено (элемент 0 всегда следует перед элементом 1), а ключи являются числовыми и последовательными. Массивы поддерживают много полезных методов, делающих этот тип данных чрезвычайно мощным средством выражения информации, которое мы рассмотрим в главе 8.

Если вы переходите на JavaScript с других языков, то вы найдете, что массивы в JavaScript — это некий гибрид эффективности индексированных массивов языка C, а также мощности динамических массивов и связанных списков. Массивы в JavaScript обладают следующими свойствами.

- Размер массива не ограничен; вы можете добавлять и удалять элементы в любое время.
- Массивы не являются гомогенными; каждый индивидуальный элемент может иметь любой тип.
- Элементы массива нумеруются от нуля. Таким образом, первый элемент в массиве — элемент 0.

Поскольку массивы — это объекты специального типа, обладающие дополнительными функциональными возможностями, вы можете присваивать массиву нечисловые (или дробные, или негативные) ключи. Хотя это и возможно, подобное противоречит основной цели массивов и может привести к непредвиденному поведению и трудно обнаруживаемым ошибкам, поэтому его лучше избегать.

Для литерала массива в JavaScript используются квадратные скобки, заключающие элементы массива, разделенные запятыми.

```
const a1 = [1, 2, 3, 4];           // массив, содержащий числа
const a2 = [1, 'two', 3, null];    // массив, содержащий смешанные типы
const a3 = [                       // массив из нескольких строк
    "What the hammer? What the chain?",
    "In what furnace was thy brain?",
    "What the anvil? What dread grasp",
    "Dare its deadly terrors clasp?",
];
const a4 = [                       // массив, содержащий объекты
    { name: "Ruby", hardness: 9 },
    { name: "Diamond", hardness: 10 },
    { name: "Topaz", hardness: 8 },
];
const a5 = [                       // массив, содержащий массивы
     [1, 3, 5],
     [2, 4, 6],
];
```

Массивы обладают свойством `length`, возвращающим количество элементов в массиве.

```
const arr = ['a', 'b', 'c'];
arr.length;                        // 3
```

Для доступа к отдельным элементам массива мы просто используем числовой индекс элемента в квадратных скобках (подобно тому, как мы обращаемся к свойствам объекта).

```
const arr = ['a', 'b', 'c'];

// получить первый элемент:
arr[0];                            // 'a'

// индекс последнего элемента в массиве - arr.length-1:
arr[arr.length - 1];               // 'c'
```

Для перезаписи значения элемента массива по определенному индексу достаточно присвоить ему новое значение[4].

[4] Обычно конструктор используется с ключевым словом new, о котором мы узнаем в главе 9; но это частный случай.

```
const arr = [1, 2, 'c', 4, 5];
arr[2] = 3;                    // теперь arr [1, 2, 3, 4, 5]
```

В главе 8 мы изучим еще множество способов изменения массивов и их содержимого.

Завершающие запятые в объектах и массивах

Внимательный читатель, возможно, уже обратил внимание, что в этих примерах кода, когда содержимое объектов и массивов охватывает несколько строк, есть *замыкающие* (или *оконечные*) запятые.

```
const arr = [
    "One",
    "Two",
    "Three",
];
const o = {
    one: 1,
    two: 2,
    three: 3,
};
```

Многие программисты их избегают, поскольку в ранних версиях Internet Explorer завершающие запятые приводили к ошибкам (даже при том, что это корректный синтаксис JavaScript). Я предпочитаю завершающие запятые, поскольку часто вырезаю и вставляю код в пределах массивов и объектов, а также добавляю код в конец объекта. Таким образом, имея завершающие запятые, мне не нужно помнить о необходимости добавить запятую в предыдущую строку; она уже там есть. Это весьма спорное соглашение, но я за него. Если вам не нравятся завершающие запятые (или их использование не принято в вашей группе), не используйте их.

Формат JavaScript Object Notation (JSON) (весьма популярный синтаксис данных, подобный JavaScript) *не допускает* завершающих запятых.

Даты

Даты и время в JavaScript представляются встроенным объектом `Date`. Это один из наиболее проблематичных аспектов языка. Первоначально это была прямая связь с Java (одна из немногих областей, в которых у JavaScript фактически есть прямое отношение к Java); с объектом `Date` может быть сложно работать, особенно если вы имеете дело с датами в различных часовых поясах.

Чтобы создать дату, инициализированную текущей датой и временем, используйте оператор `new Date()`.

```
const now = new Date();
now; // пример: Thu Aug 20 2015 18:31:26 GMT-0700 (Pacific Daylight Time)
```

Можно создать дату, инициализированную определенным днем (в 12:00 ночи).

```
const halloween = new Date(2016, 9, 31); // обратите внимание: месяцы
                                         // отсчитываются от
                                         // нуля: 9=October
```

Можно создать дату, инициализированную определенными датой и временем.

```
const halloweenParty = new Date(2016, 9, 31, 19, 0); // 19:00 = 7:00 pm
```

Имея объект даты, можно получить его компоненты.

```
halloweenParty.getFullYear();      // 2016
halloweenParty.getMonth();         // 9
halloweenParty.getDate();          // 31
halloweenParty.getDay();           // 1 (Mon; 0=Sun, 1=Mon,...)
halloweenParty.getHours();         // 19
halloweenParty.getMinutes();       // 0
halloweenParty.getSeconds();       // 0
halloweenParty.getMilliseconds(); // 0
```

Подробно мы будем рассматривать даты в главе 15.

Регулярные выражения

Регулярное выражение (regular expression или regex, или regexp) является неким подмножеством языка JavaScript. Это модификация общего языка, предоставляемая многими языками программирования и представляющая компактный способ выполнения операций сложного поиска и замены в строках. Регулярные выражения рассматриваются в главе 17. Регулярные выражения в JavaScript представляются объектом `RegExp`, и его литеральный синтаксис состоит из символов между парой косых черт. Вот несколько примеров (выглядящих бессмысленно, если вы никогда не видели регулярных выражений прежде).

```
// чрезвычайно простое средство распознавания адресов электронной почты
const email = /\b[a-z0-9._-]+@[a-z_-]+(?:\.[a-z]+)+\b/;

// распознавание номера телефона в США
const phone = /(:?\+1)?(:?\(\d{3}\)\s?|\d{3}[\s-]?)\d{3}[\s-]?\d{4}/;
```

Отображения и наборы

В спецификацию ES6 введены типы данных `Map` и `Set`, а также их "слабые" дубликаты, `WeakMap` и `WeakSet`. Отображения, подобно объектам, сопоставляют ключи

со значениями, но обладают по сравнению с объектами некоторыми преимуществами в определенных ситуациях. Наборы подобны массивам, но не могут содержать дубликатов. Их слабые аналоги обладают подобными функциями, но более эффективны в определенных ситуациях, в обмен на функциональные возможности.

Мы рассмотрим отображения и наборы в главе 10.

Преобразование типов данных

Преобразование данных из одного типа в другой — весьма популярная задача. Данные, вводимые пользователем или поступающие из других систем, зачастую следует преобразовывать. В этом разделе рассматриваются некоторые из наиболее общих методик преобразования данных.

Преобразование в числовой формат

Весьма распространено преобразование строк в числа. Когда вы получаете ввод от пользователя, то обычно это строка, даже если вы запрашивали числовое значение. JavaScript предоставляет несколько методов преобразования строки в число. Первый из них — использовать конструктор объекта `Number`[5].

```
const numStr = "33.3";
const num = Number(numStr); // это создает значение числа, а *не*
                            // экземпляр объекта Number
```

Если строка не может быть преобразована в число, возвращается `NaN`.

Второй метод подразумевает использование встроенной функции `parseInt` или `parseFloat`. Они ведут себя аналогично конструктору `Number`, за несколькими исключениями. Функция `parseInt` позволяет определить *основание системы счисления* (radix) получаемого значения. Например, она позволяет задать основание 16, чтобы получать шестнадцатеричные числа. Основание системы счисления рекомендуется определять всегда, даже если это 10 (стандартное). Функции `parseInt` и `parseFloat` отбрасывают все, что находят после числа, позволяя использовать не полностью отфильтрованный ввод. Вот примеры.

```
const a = parseInt("16 volts", 10); // слово " volts" игнорируется; 16
                                    // преобразуется по основанию 10
const b = parseInt("3a", 16);       // преобразуется шестнадцатеричное
                                    // 3a; результат — 58
const c = parseFloat("15.5 kph");   // слово " kph" игнорируется;
                                    // parseFloat всегда подразумевает
                                    // основание 10
```

[5] Обычно конструктор используется с ключевым словом new, о котором мы узнаем в главе 9; но это частный случай.

Используя метод `valueOf()`, объект `Date` можно преобразовать в число, представляющее количество миллисекунд с полуночи 1 января 1970 года (UTC).

```
const d = new Date();    // текущая дата
const ts = d.valueOf(); // числовое значение: миллисекунды с
                        // полуночи 1 января 1970 года (UTC)
```

Иногда имеет смысл преобразовать логическое значение в 1 (истина) или 0 (ложь). Для преобразования используется условный оператор (о котором мы узнаем в главе 5).

```
const b = true;
const n = b ? 1 : 0;
```

Преобразование в строку

У всех объектов в JavaScript есть метод `toString()`, возвращающий строковое представление объекта. Его стандартная реализация не особенно полезна. Она хороша для чисел, хотя преобразовывать числа в строку приходится не часто: это преобразование обычно происходит автоматически во время конкатенации или интерполяции строк. Но если вам когда-либо понадобится преобразовать число в строку, то метод `toString()` — это то, что нужно.

```
const n = 33.5;
n;                       // 33.5 - число
const s = n.toString();
s;                       // "33.5" - строка
```

Объекты `Date` обладают достаточно полезной реализацией метода `toString()`, но у большинства объектов она просто возвращает строку `"[object Object]"`. Объекты могут быть изменены так, чтобы возвращать более полезное строковое представление, но это тема главы 9. Метод `toString()` массива преобразует в строки каждый элемент этого массива, а затем объединяет их в одну строку, разделив запятыми.

```
const arr = [1, true, "hello"];
arr.toString();          // "1,true,hello"
```

Преобразование в логическое значение

В главе 5 мы узнаем о концепции “истинного” и “ложного” значения в JavaScript, а также о способах приведения всех значений к истине или лжи, поэтому мы не будем рассматривать все эти подробности здесь. Однако стоит упомянуть, что использование оператора “не” (`!`) дважды позволяет преобразовать любое значение в логическое. При его однократном использовании значение преобразуется в логическое, но это противоположность тому, что нужно; при его повторном использовании значение преобразуется в то, что нужно. Подобно числовым преобразованиям вы

можете также использовать конструктор `Boolean` (снова без ключевого слова `new`) для достижения того же результата.

```
const n = 0;           // значение "false"
const b1 = !!n;        // false
const b2 = Boolean(n); // false
```

Заключение

Типы данных, доступные в языке программирования, являются вашими стандартными строительными блоками для того, что вы можете выразить на языке. Из этой главы мы узнали следующие ключевые пункты, применимые для повседневного программирования.

- В JavaScript есть шесть базовых типов (строка, число, логическое значение, пустое значение, неопределенное значение и символ) и тип объекта.
- Все числа в JavaScript — это числа с плавающей точкой двойной точности.
- Массивы — это специальные типы объектов; наряду с объектами они являются очень мощным и гибким типом данных.
- Другие часто используемые типы данных (даты, отображения, наборы и регулярные выражения) являются объектами специальных типов.

Вы, вероятнее всего, будете использовать строки довольно часто, и я настоятельно рекомендую усвоить правила построения строк и работу строковых шаблонов.

Управление потоком

Общий совет начинающим программистам — *следуйте рецептам*. Этот совет может быть полезен, но у него есть один прискорбный недостаток: для достижения воспроизводимых результатов на кухне нужно *минимизировать* количество возможных альтернатив. Как только рецепт составлен, необходимо следовать ему шаг за шагом без отклонений. Иногда, конечно, альтернативы будут: "вместо масла годится жир" или "соль по вкусу", но рецепт, прежде всего, является списком этапов, которым необходимо следовать.

Вся эта глава — об *альтернативах* и их *выборе*, позволяющем вашей программе реагировать на изменяющиеся условия и осмысленно выполнять повторяющиеся задачи.

Если у вас уже есть опыт программирования, особенно на языке с синтаксисом, унаследованным от C (C++, Java, C#), и вы знакомы с операторами управления потоком, то вы можете просмотреть или смело пропустить первую часть этой главы. Но если вы сделаете это, то ничего не узнаете об играх, популярных среди моряков в XIX веке.

Учебник для новичков в управлении потоком

Концепцию выбора из возможных альтернатив отражает *блок-схема* (flowchart), являющаяся визуальным представлением потока выполнения программы. В качестве примера этой главы мы собираемся написать *модель* (simulation). В частности, мы собираемся смоделировать популярную среди гардемаринов Королевского флота середины XIX века игру *Корона и Якорь* (Crown and Anchor).

Игра проста: есть доска с шестью квадратами, помеченными символами "Корона", "Якорь", "Бубны", "Пики", "Трефы" и "Червы". Моряк ставит любое количество монет на любую комбинацию полей — это ставки. Затем он[1] бросает три шестигранные игральные кости с такими же изображениями, как и на полях на доске. Поскольку каждое выпавшее значение соответствует квадрату, поставивший на него матрос выигрывает определенную сумму денег. Ниже приведены примеры выплат при выигрыше.

[1] Вопреки моему желанию пришлось использовать пример, связанный с дискриминацией по половому признаку, ведь женщины не служили в Королевском флоте до 1917 года.

Ставка, пенсов	Выпало	Выплата, пенсов
5 на Корону	Корона, Корона, Корона	15
5 на Корону	Корона, Корона, Якорь	10
5 на Корону	Корона, Черва, Пика	5
5 на Корону	Черва, Якорь, Пика	0
3 на Корону, 2 на Пики	Корона, Корона, Корона	9
3 на Корону, 2 на Пики	Корона, Пика, Якорь	5
По 1 на все поля	Любые	3 (не лучшая стратегия!)

Я выбрал этот пример потому, что он не слишком сложен и при небольшом количестве воображения демонстрирует основные операторы управления потоком. Хотя и маловероятно, что вам когда-либо придется моделировать игорные предпочтения моряков XIX столетия, подобный тип моделей весьма распространен во многих приложениях. В случае данной игры, возможно, понадобится построить математическую модель, чтобы определить, должны ли выпасть Корона и Якорь, чтобы выплатить деньги или перейти к следующему событию игры. Создаваемая в этой главе модель применяется для подтверждения правильности нашего выбора.

Сама игра проста, но есть много тысяч способов, которыми она может начинаться. Наш моряк, давайте назовем его "Томас" (хорошее солидное британское имя), сначала будет очень обобщенным, но со временем его поведение станет более детализированным.

Давайте начнем с основ: условий начала и остановки. При каждом увольнении на берег Томас получает по 50 пенсов и тратит их на *Корону и Якорь*. У Томаса есть лимит: если повезет, его деньги удвоятся и он уйдет по крайней мере со 100 пенсами в кармане (примерно половина его месячной зарплаты). Если он не удвоит свои деньги, то проиграет их до конца.

Давайте разделим игру на три этапа: размещение ставок, бросок костей и получение выигрыша (если он есть). Теперь, имея очень простое, общее представление о поведении Томаса, мы можем составить описывающую его блок-схему, представленную на рис. 4.1.

На блок-схеме ромбами обозначены *решения* "да или нет", а прямоугольники представляют *действия*. Мы используем круги для обозначения начала и конца.

Блок-схема (как мы ее нарисовали) еще не готова для превращения непосредственно в программу. Изложенные здесь этапы понятны для человека, но слишком сложны для компьютеров. Например, "Бросок костей" вовсе не очевиден для компьютера. Что такое игральная кость? Как вы ее бросаете? Для решения этой задачи этапы "размещение ставок", "бросок костей" и "получение выигрыша" требуют *собственных* блок-схем (я обозначил это на блок-схеме, закрасив соответствующие действиям прямоугольники). Если у вас есть достаточно большой лист бумаги, то можете поместить их все вместе, но в этой книге мы представим их по отдельности.

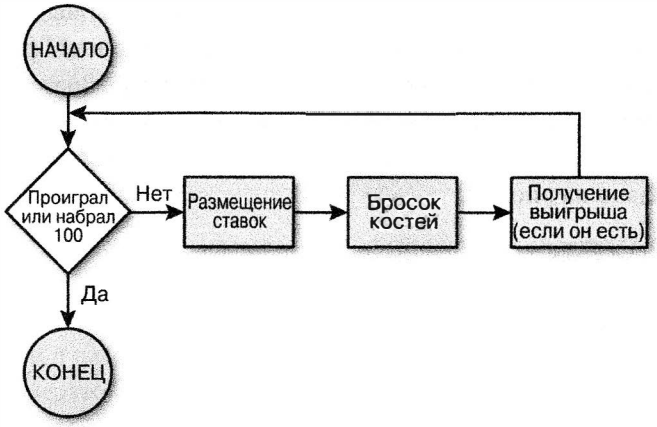


Рис. 4.1. Блок-схема модели игры "Корона и Якорь"

Кроме того, наш блок принятия решения слишком неопределенен для компьютера: "проиграл или набрал 100 пенсов?" — это не то, что компьютер может понять. Так что же *может* понять компьютер? В этой главе мы ограничим действия своей блок-схемы следующими.

- Присваивание значения переменным: `funds = 50`, `bets = {}`, `hand = []`.
- Случайное целое число от *m* до *n* включительно: `rand(1, 6)` (это вспомогательная функция, мы рассмотрим ее позже).
- Строка случайно выпавших граней ("черва", "корона" и т.д.): `randFace()` (другая вспомогательная функция).
- Присваивание свойств объекта: `bets["heart"] = 5`, `bets[randFace()] = 5`.
- Добавление элементов в массив: `hand.push(randFace())`.
- Простая арифметика: `funds - totalBet`, `funds + winnings`.
- Инкремент: `roll++` (это общепринятое сокращение, означающее "добавить к переменной `roll` единицу").

И мы ограничим свои решения на блок-схемах следующими.

- Сравнение чисел (`funds > 0`, `funds < 100`).
- Сравнение на равенство (`totalBet === 7`; о том, почему используются три знака равенства, мы узнаем в главе 5).
- Логический оператор (`funds > 0 && funds < 100`; два амперсанда означают "и", как мы узнаем в главе 5).

Все эти "разрешенные действия" являются действиями, которые мы можем написать в JavaScript с небольшой интерпретацией или трансляцией или вообще без нее.

Еще одно обсуждение терминологии в этой главе: мы будем использовать термины *истинно* (truthy) и *ложно* (falsy). Это не просто упрощенные или "симпатичные" версии *истины* (true) и *лжи* (false): у них есть значение в JavaScript. Значение этих терминов объясняется в главе 5, а пока вы можете считать их эквивалентами "истины" и "лжи".

Теперь, когда ограничения языка известны, мы можем переделать свою блок-схему, как показано на рис. 4.2.

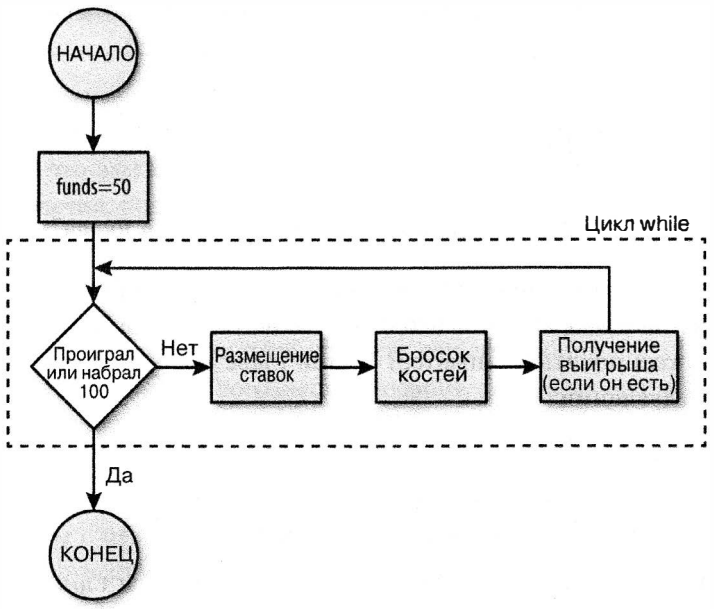


Рис. 4.2. Блок-схема модели игры "Корона и Якорь" (усовершенствованная)

Циклы `while`

Теперь у нас, наконец, есть нечто, что мы можем преобразовать непосредственно в код. В нашей блок-схеме уже есть первый оператор управления потоком, который мы будем обсуждать; это цикл `while`. Цикл `while` повторяет код, пока выполняется его *условие* (condition). Условие в нашей блок-схеме — это `funds > 1 && funds < 100`. Давайте посмотрим, как это выглядит в коде.

```
let funds = 50;     // Начальное условие

while(funds > 1 && funds < 100) {
    // Размещение ставок

    // Бросок костей

    // Получение выигрыша
}
```

Если запустить эту программу как есть, то она будет выполняться бесконечно, поскольку, начав с 50 пенсов, наличная сумма никогда не увеличится и не уменьшится, поэтому условие всегда останется истинным. Тем не менее, прежде чем приступать к подробностям, необходимо поговорить о *блоках операторов* (block statement).

Блоки операторов

Блоки операторов (иногда называемые *составными операторами* (compound statement)) не являются операторами управления потоком выполнения, но идут взявшись с ними за руки. Блок операторов — это только набор операторов, заключенных в фигурные скобки и рассматриваемый JavaScript как единый блок. Хотя вполне можно создать блок операторов сам по себе, ни смысла, ни удобства в этом особого нет.

```
{   // начало блока операторов
    console.log("statement 1");
    console.log("statement 2");
} // конец блока операторов

console.log("statement 3");
```

Первые два вызова `console.log` находятся в блоке; это бессмысленно, но вполне допустимо.

Блоки операторов становятся полезными совместно с операторами управления потоком. Например, в цикле оператора `while` выполняется *весь блок операторов*, и только затем проверяется условие снова. Например, если бы мы хотели сделать "два шага вперед и шаг назад", то мы могли бы написать

```
let funds = 50;         // Начальное условие

while(funds > 1 && funds < 100) {

    funds = funds + 2; // два шага вперед
    funds = funds - 1; // один шаг назад
}
```

Этот цикл `while` в конечном счете закончится: за каждый цикл значение `funds` увеличивается на два и уменьшения на единицу. В конечном счете значение `funds` составит 100, и цикл закончится.

Использование блока операторов при управлении потоком очень распространено, но не обязательно. Например, если мы хотим просто посчитать до 100 парами, то блок операторов не обязателен.

```
let funds = 50;     // Начальное условие

while(funds > 1 && funds < 100)
    funds = funds + 2;
```

Отступ

По большей части дополнительный отступ (включая символ перевода строки[2]): не влияет на работу интерпретатора JavaScript: один пробел так же хорош, как и 10, а 10 пробелов — как и 10 новых строк. Это не означает, что вы можете использовать отступ как попало. Например, приведенный выше оператор `while` эквивалентен

```
while(funds > 1 && funds < 100)

funds = funds + 2;
```

Однако так взаимоотношения этих двух операторов не вполне очевидны! Такое форматирование вводит в заблуждение, и его следует избегать. Однако следующие эквиваленты весьма распространены, а кроме того, они более однозначны.

```
// без новой строки
while(funds > 1 && funds < 100) funds = funds + 2;

// без новой строки, блок с одним оператором
while(funds > 1 && funds < 100) { funds = funds + 2; }
```

Некоторые люди настаивают, чтобы тела оператора управления потоком (для единообразия и ясности) всегда были блоком операторов (даже если они содержат только один оператор). Хотя я и не принадлежу к этому лагерю, я должен указать, что небрежные отступы иногда сильно вводят в заблуждение.

```
while(funds > 1 && funds < 100)
    funds = funds + 2;
    funds = funds - 1;
```

На первый взгляд, в теле цикла `while` выполняется два оператора (два шага вперед и один назад), но поскольку никакого блока операторов здесь нет, JavaScript интерпретирует это так.

```
while(funds > 1 && funds < 100)
    funds = funds + 2; // тело цикла while

funds = funds - 1;     // после цикла while
```

Я сторонник тех, кто говорит, что отсутствие блоков для тел с одиночным оператором вполне допустимо, но, конечно, вы всегда должны нести ответственность за отступ, чтобы ясно выражать свое намерение. Кроме того, работая в группе или над проектом реализации с открытым исходным кодом, вы должны придерживаться всех принятых группой стилистических правил независимо от ваших личных предпочтений.

[2] Новые строки после оператора `return` приводят к проблемам; дополнительная информация приведена в главе 6.

Хотя по вопросу об использовании блоков для тел с одиночным оператором есть разногласия, одно синтаксическое правило неоспоримо: смешение блоков и одиночных операторов в том же операторе `if` недопустимо.

```
// не делайте так
if(funds > 1) {
    console.log("There's money left!");
    console.log("That means keep playing!");
} else
    console.log("I'm broke! Time to quit.");
// или так
if(funds > 1)
    console.log("There's money left! Keep playing!");
else {
    console.log("I'm broke"!);
    console.log("Time to quit.")
}
```

Вспомогательные функции

Для примеров этой главы нам понадобятся две *вспомогательные функции*. Мы еще не рассматривали функции (и генераторы псевдослучайных чисел), это темы следующих глав; а пока скопируйте эти две вспомогательные функции дословно.

```
// возвращает случайное целое число в диапазоне [m, n] (включительно)
function rand(m, n) {
    return m + Math.floor((n - m + 1)*Math.random());
}

// случайно возвращает строку, представляющую одну из шести
// граней Короны и Якоря
function randFace() {
    return ["crown", "anchor", "heart", "spade", "club", "diamond"]
        [rand(0, 5)];
}
```

Оператор `if...else`

Давайте заполним теперь один из закрашенных прямоугольников "Размещение ставок" нашей блок-схемы. Так как Томас делает ставки? У него есть целый ритуал. Он достает из своего правого кармана случайную горсть монет (от только одной монеты до всех). Это будет его наличность для данного раунда. Томас суеверен, он полагает, что число 7 приносит удачу. Так, случайно вытащив 7 пенсов, он возвращается в карман и ставит *все* свои деньги на поле "Червы". В противном случае он ставит на случайные поля (это тоже пока отложим). Давайте рассмотрим блок-схему "Размещение ставок" на рис. 4.3.

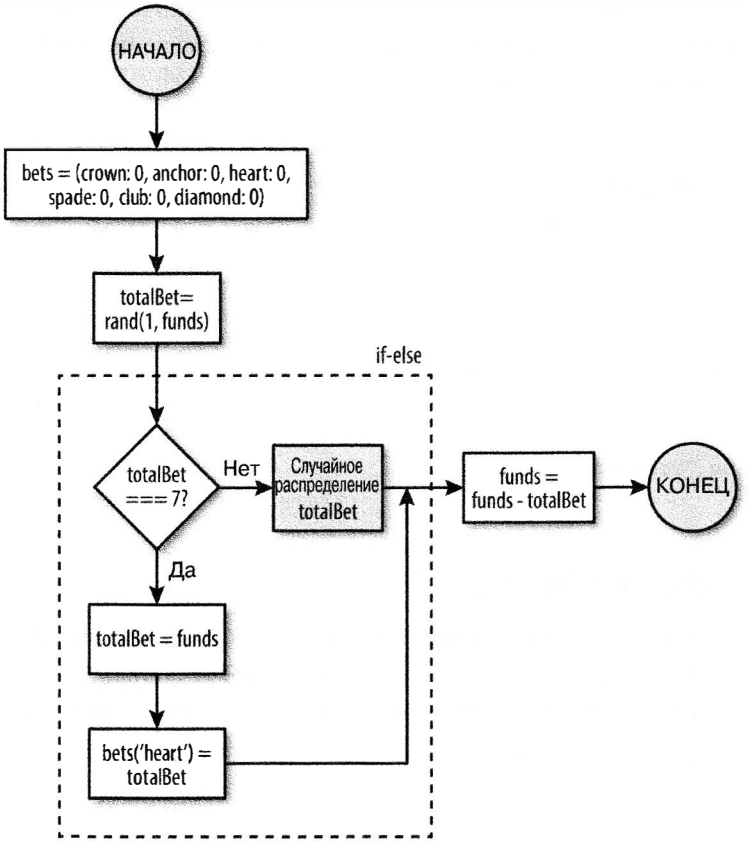


Рис. 4.3. Модель игры "Корона и Якорь": блок-схема размещения ставок

Блок принятия решения в середине (`totalBet === 7`) здесь представлен оператором `if...else`. Обратите внимание, что в отличие от оператора `while` он не возвращается циклически к себе самому: решение принимается, а затем выполнение продолжается. Преобразуем эту блок-схему в код JavaScript.

```
const bets = { crown: 0, anchor: 0, heart: 0,
    spade: 0, club: 0, diamond: 0 };
let totalBet = rand(1, funds);
if(totalBet === 7) {
    totalBet = funds;
    bets.heart = totalBet;
} else {
    // Распределение всех ставок
}
funds = funds - totalBet;
```

Далее мы увидим, что часть `else` оператора `if...else` необязательна.

Цикл do...while

Когда Томас не вытаскивает случайно 7 пенсов, он распределяет наличность между полями случайно. У него и для этого есть ритуал: он держит монеты в правой руке, а левой выбирает их случайное количество (от только одной до всех) и ставит на случайное поле (иногда на то же поле несколько раз). Теперь мы можем модифицировать свою блок-схему так, чтобы отразить это случайное распределение всех ставок, как показано на рис. 4.4.

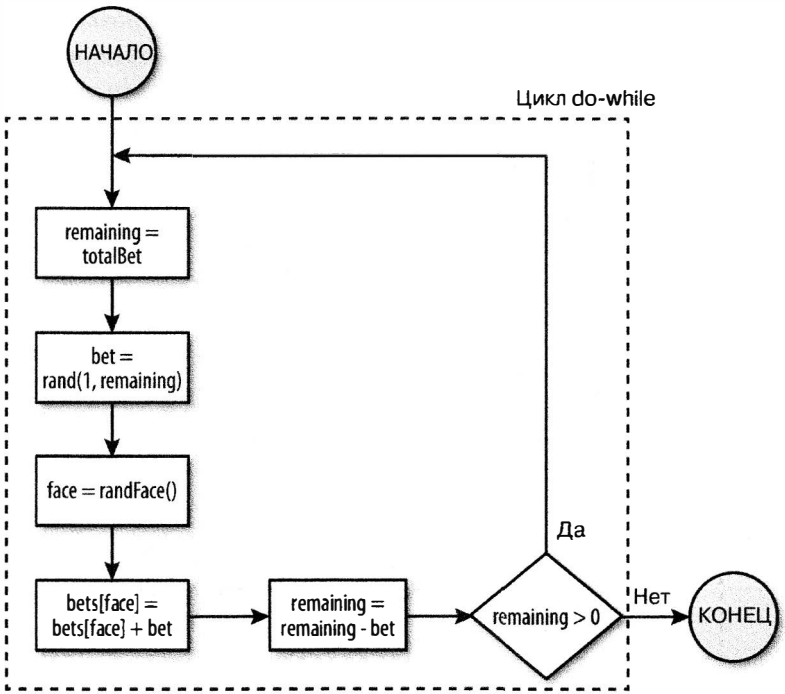


Рис. 4.4. Модель игры "Корона и Якорь": блок-схема распределения ставок

Обратите внимание, насколько это отличается от цикла `while`: решение принимается в конце, а не в начале. Цикл `do...while` подходит тогда, когда вы знаете, что тело цикла необходимо выполнить *по крайней мере один раз* (если условие цикла `while` изначально ложно, он не будет выполнен даже однажды). Вот как это выглядит в JavaScript.

```
let remaining = totalBet;
do {
    let bet = rand(1, remaining);
    let face = randFace();
    bets[face] = bets[face] + bet;
    remaining = remaining - bet;
} while(remaining > 0);
```

Цикл `for`

Теперь Томас сделал все ставки! Время бросать кости.

Цикл `for` чрезвычайно гибок (он может даже заменить цикл `while` или `do... while`), но он лучше всего подходит для случая, когда необходимо фиксированное количество циклов (особенно когда необходимо знать номер текущего цикла), что делает его идеальным для броска фиксированного количества игральных костей (в данном случае — трех). Давайте приступим к "блок-схеме" броска игральной кости, представленной на рис. 4.5.

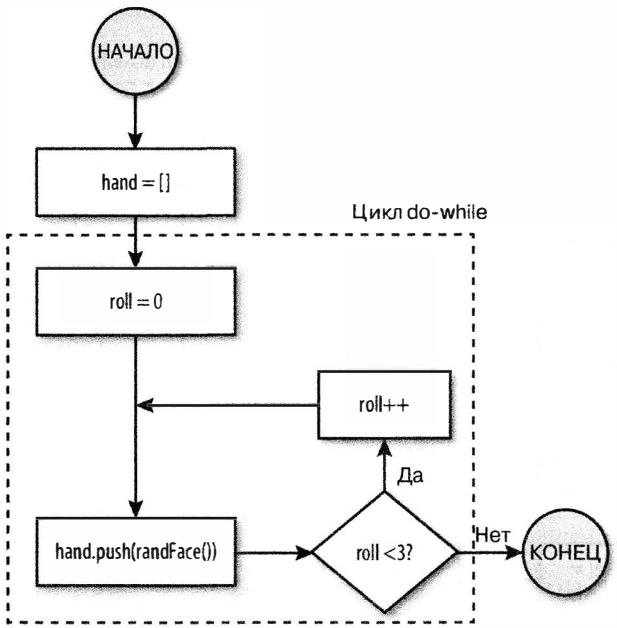


Рис. 4.5. Модель игры "Корона и Якорь": блок-схема броска кости

Цикл `for` состоит из трех частей: *инициализация* (`roll = 0`), *условие* (`roll < 3`) и *заключительное выражение* (`roll++`). Это нечто, что не может быть создано с использованием цикла `while`, здесь вся информация цикла удобно размещается в одном месте. Вот как это выглядит в JavaScript.

```
const hand = [];
for(let roll = 0; roll < 3; roll++) {
    hand.push(randFace());
}
```

Программисты предпочитают отсчитывать все от 0, вот почему мы начинаем с броска 0 и останавливаемся на броске 2.

Общепринято соглашение использовать переменную `i` (сокращение от "индекс") в цикле `for` независимо от того, что вы подсчитываете, хотя на самом деле можно использовать любое имя переменной. Здесь я решил выбрать имя `roll`, чтобы дать ясно понять, что мы подсчитываем количество бросков, но, честно говоря, когда я писал этот пример впервые, я использовал имя `i` по привычке!

Оператор `if`

Мы почти закончили! Все ставки сделаны и кости брошены; остается лишь получить выигрыш, какой есть. В массиве `hand` содержатся три случайно выпавшие грани, поэтому мы используем другой цикл `for` для выяснения, есть ли выигрыш. Для этого мы используем оператор `if` (на сей раз без директивы `else`). Наша заключительная блок-схема представлена на рис. 4.6.

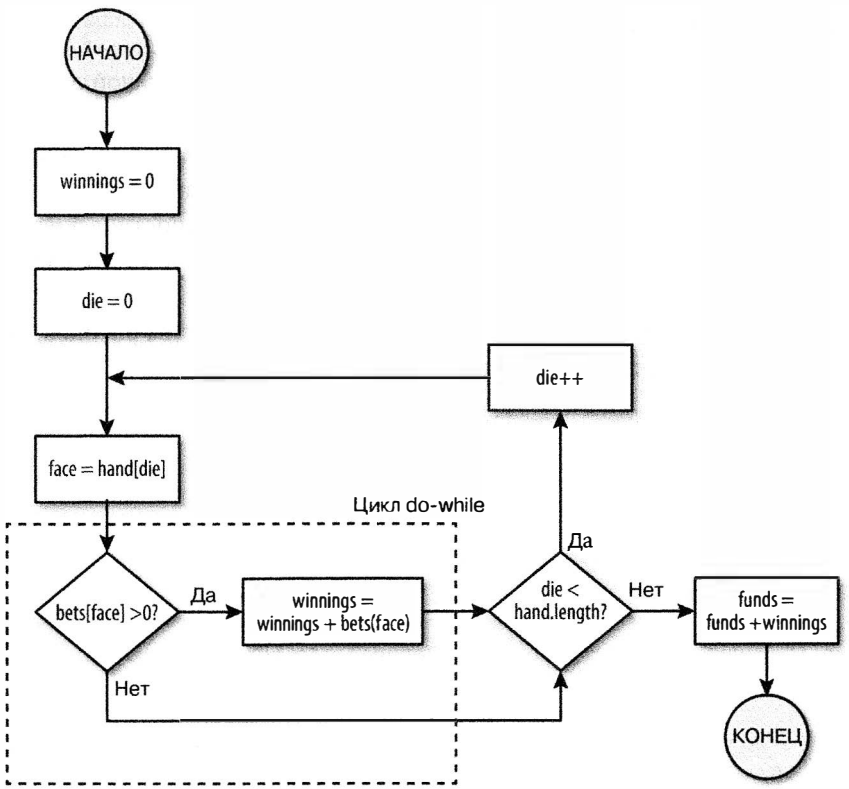


Рис. 4.6. Модель игры "Корона и Якорь": блок-схема получения выигрыша

Обратите внимание на различие между оператором `if...else` и оператором `if`: только одна из его ветвей приводит к действию, тогда как у оператора `if...else` — обе. Мы преобразуем это в код для заключительной части задачи.

```
let winnings = 0;
for(let die=0; die < hand.length; die++) {
    let face = hand[die];
    if(bets[face] > 0) winnings = winnings + bets[face];
}
funds = funds + winnings;
```

Обратите внимание, что вместо счета до 3 в цикле `for` мы считаем до `hand.length` (что также дает 3). Задача этой части программы — вычислить выигрыш для *любого расклада*. Хотя правила игры требуют набора из трех игральных костей, правила могли измениться... или, возможно, появятся дополнительные игральные кости в качестве бонуса или меньше игральных костей в качестве штрафа. Дело в том, что совсем не сложно сделать этот код более *обобщенным*. Если правила позволят изменять количество игральных костей, то нам не придется заботиться об изменении этого кода: он будет работать правильно независимо от количества костей в игре.

Объединим все вместе

Чтобы собрать все части блок-схемы вместе, понадобится большой лист бумаги, но мы можем написать целую программу сравнительно легко.

В приведенном ниже коде программы (включающем вспомогательные функции) есть также несколько вызовов `console.log`, так вы сможете наблюдать за игрой Томаса (не волнуйтесь о понимании работы системы вывода диагностических сообщений: она использует некоторые дополнительные методики, которые мы рассмотрим в последующих главах).

Мы также добавим переменную `round` для подсчета количества раундов, которые сыграл Томас, исключительно для демонстрации.

```
// возвращает случайное целое число в диапазоне [m, n] (включительно)
function rand(m, n) {
    return m + Math.floor((n - m + 1)*Math.random());
}

// случайно возвращает строку, представляющую одну из шести
// граней Короны и Якоря
function randFace() {
    return ["crown", "anchor", "heart", "spade", "club", "diamond"]
         [rand(0, 5)];
}

let funds = 50; // Начальные условия
let round = 0;

while(funds > 1 && funds < 100) {
    round++;
    console.log('round ${round}:');
    console.log('\tstarting funds: ${funds}p');
```

```
    // Размещение ставок
    let bets = { crown: 0, anchor: 0, heart: 0,
        spade: 0, club: 0, diamond: 0 };
    let totalBet = rand(1, funds);
    if(totalBet === 7) {
        totalBet = funds;
        bets.heart = totalBet;
    } else {
        // Распределение всех ставок
        let remaining = totalBet;
        do {
            let bet = rand(1, remaining);
            let face = randFace();
            bets[face] = bets[face] + bet;
            remaining = remaining - bet;
        } while(remaining > 0)
    }
    funds = funds - totalBet;
    console.log('\tbets: ' +
    Object.keys(bets).map(face => '${face}: ${bets[face]} pence').join(', ') +
    ' (total: ${totalBet} pence)');

    // Бросок костей
    const hand = [];
    for(let roll = 0; roll < 3; roll++) {
        hand.push(randFace());
    }

    console.log('\thand: ${hand.join(', ')}');

    // Получение выигрыша
    let winnings = 0;
    for(let die=0; die < hand.length; die++) {
        let face = hand[die];
        if(bets[face] > 0) winnings = winnings + bets[face];
    }
    funds = funds + winnings;
    console.log('\twinnings: ${winnings}');
}
console.log('\tending funs: ${funds}');
```

Операторы управления потоком в JavaScript

Теперь, когда у нас есть четкое понимание того, что именно *делают* операторы управления потоком, и некое представление о большинстве их базовых возможностей, мы можем перейти к подробностям.

Мы также собираемся оставить блок-схемы. Это прекрасный инструмент визуализации (особенно для тех, кто лучше воспринимает информацию визуально), но они слишком громоздки.

По правде говоря, операторы управления потоком можно разделить на две категории: *условные выражения* (conditional), они же *ветвления* (branching), и *циклы* (loop). Условные выражения для управления потоком выполнения (`if` и `if...else`, которые мы уже видели, и `switch`, который мы рассмотрим вскоре) обеспечивают ветвление пути: если есть два или больше путей, то они обеспечивают выбор. Операторы цикла (`while`, `do...while` и `for`) повторяют выполнение тела операторов, пока соблюдается условие.

Исключения в управлении потоком

Есть четыре оператора, позволяющие изменить нормальный ход управления потоком. Можете считать их "козырными картами" управления потоком.

- Оператор `break`. Немедленно прерывает цикл.
- Оператор `continue`. Переходит к следующему шагу в цикле.
- Оператор `return`. Осуществляет выход из текущей функции (независимо блока, в котором он находится). (См. главу 6.)
- Оператор `throw`. Вызывает *исключение* в программе (exception), которое следует обработать обработчиком исключений (он может находиться за пределами текущего оператора управления потоком). (См. главу 11.)

Использование операторов станет понятнее впоследствии; а пока достаточно знать, что эти четыре оператора способны изменить поведение конструкций управления потоком.

Сцепление операторов `if...else`

Фактически сцепление операторов `if...else` — это не специальный синтаксис, а просто серия операторов `if...else`, в которой каждая директива `else` содержит другой оператор `if...else`. Это вполне обычная схема, заслуживающая упоминания. Например, если суеверие Томаса начнет простираться на дни недели и он начнет ставить в среду только по одному пенни, мы могли бы выразить эту логику в виде цепочки операторов `if...else`.

```
if(new Date().getDay() === 3) {      // new Date().getDay() возвращает
                                     // текущий день
    totalBet = 1;          // номер дня недели, где 0 = воскресенье
} else if(funds === 7) {
    totalBet = funds;
} else {
    console.log("Здесь нет никаких суеверий!");
}
```

Объединив операторы `if...else` таким способом, мы создали выбор одного пути из трех вместо двух. Внимательный читатель мог бы заметить, что технически мы отклонились от установленного правила (не смешивать одиночные операторы и блоки операторов), но это исключение из правила: таков общепринятый шаблон и он не затрудняет чтения. Мы могли бы переписать эти блоки операторов так.

```
if(new Date().getDay() === 3) {
    totalBet = 1;
} else {
    if(funds === 7) {
        totalBet = funds;
    } else {
        console.log("Здесь нет никаких суеверий!");
    }
}
```

Мы сделали свой код более правильным, объемным и подробным, но ясности не получили.

Метасинтаксис

Термин *метасинтаксис* (metasyntax) означает синтаксис, описывающий еще один синтаксис. Обладающие подготовкой в информатике сразу вспомнят о *расширенной форме Бэкуса–Наура* (Extended Backus-Naur Form — EBNF) — простой концепции с невероятно пугающим названием.

В остальной части этой главы я буду использовать метасинтаксис для краткого описания синтаксиса управления потоком JavaScript. Используемый мной метасинтаксис прост, неформален и, что важнее всего, применяется для документации JavaScript в сети *Mozilla Developer Network* (MDN). Поскольку MDN — это ресурс, который вы, несомненно, будете использовать очень часто, знакомство с ним будет полезным.

В этом метасинтаксисе есть только два реальных элемента: нечто, окруженное квадратными скобками, является *необязательным*, а многоточие (технически — троеточие) означает "здесь есть что-то еще". Слова используются как знакоместа, а их значение ясно из контекста. Например, *`оператор1`* и *`оператор2`* представляют два разных оператора, *`выражение`* — это нечто, возвращающее значение, а *`условие`* подразумевает выражение, возвращающее истину или ложь.

Помните, что блок операторов — это оператор..., поэтому везде, где можно использовать оператор, можно использовать и блок операторов.

Поскольку мы уже знакомы с некоторыми операторами управления потоком, давайте рассмотрим их метасинтаксис.

Оператор `while`

```
while(условие)
    оператор
```

Пока *условие* истинно, *оператор* будет выполняться.

Оператор `if...else`

```
if(условие)
    оператор1
[else
    оператор2]
```

Если *условие* истинно, будет выполняться *оператор1*; в противном случае будет выполняться *оператор2* (с учетом наличия части `else`).

Оператор `do...while`

```
do
    оператор
while(условие);
```

оператор выполнится по крайней мере однажды и будет выполняться, пока *условие* истинно.

Оператор `for`

```
for([инициализация]; [условие]; [заключительное_выражение])
    оператор
```

Перед началом выполнения цикла осуществляется *инициализация*. Если *условие* истинно, выполняется *оператор*, а затем выполняется *заключительное_выражение*, прежде чем снова проверять *условие*.

Дополнительные шаблоны цикла `for`

При использовании оператора запятой (о котором речь пойдет в главе 5), мы можем объединить несколько операторов присваивания в инициализации и заключительном выражении. Вот пример цикла `for` для вывода первых восьми чисел Фибоначчи.

```
for(let temp, i=0, j=1; j<30; temp = i, i = j, j = i + temp)
    console.log(j);
```

В этом примере мы объявляем несколько переменных (`temp`, `i` и `j`), а затем изменяем значение каждой из них в заключительном выражении. Подобно тому как использование оператора запятой позволяет сделать в цикле `for` больше, полное отсутствие операторов позволяет сделать цикл бесконечным.

```
for(;;) console.log("Я буду работать вечно!");
```

В этом цикле for условие undefined, что соответствует лжи, а значит, у цикла никогда не будет причины завершиться.

Как правило в обычных циклах for выполняется инкремент или декремент целочисленного индекса, но это не обязательно, работать будет любое выражение. Вот несколько примеров.

```
let s = '3';                        // строка, содержащая число
for(; s.length<10; s = ' ' + s); // дополненная нулями строка; обратите
                                    // внимание, что нужно включить точку с
                                    // запятой, чтобы закончить цикл for!

for(let x=0.2; x<3.0; x += 0.2)  // инкремент не целого числа
    console.log(x);

for(; !player.isBroke;)       // использование свойства объекта в условии
    console.log("Все еще в игре!");
```

Обратите внимание, что цикл for всегда можно переписать как цикл while. Другими словами,

```
for([инициализация]; [условие]; [заключительное_выражение])
    оператор
```

эквивалентно

```
[инициализация]
while([условие]) {
    оператор
    [заключительное_выражение]
}
```

Однако тот факт, что цикл for можно переписать как цикл while, вовсе не означает, что это нужно делать. Преимущество цикла for в том, что вся информация для управления циклом находится тут же, в первой строке, делая происходящее вполне очевидным. Кроме того, инициализация переменных с ключевым словом let в цикле for ограничивает их область видимости телом цикла (больше об этом — в главе 7); если преобразовать такой оператор for в оператор while, то управляющие переменные при необходимости будут доступны за пределами тела цикла for.

Операторы switch

В то время как операторы if...else позволяют выбрать один из двух путей, операторы switch позволяют выбрать один путь из нескольких на основании единого условия. Для этого условие должно быть чем-то бо́льшим, чем значение "истина/ложь": условие оператора switch — это *выражение*, вычисление которого дает значение. Синтаксис оператора switch таков.

```
switch(выражение) {
    case значение1:
```

```
        // выполняется, когда результат выражения соответствует значение1
        [break;]
    case значение2:
        // выполняется, когда результат выражения соответствует значение2
        [break;]
        ...
    case значениеN:
        // выполняется, когда результат выражения соответствует значениеN
        [break;]
    default:
        // выполняется, когда ни одно из значений не соответствует
        // значению выражения
        [break;]
}
```

JavaScript вычисляет *выражение*, выбирает первый соответствующий раздел `case` и выполняет его операторы, пока не встретит оператор `break`, `return`, `continue`, `throw` или конец оператора `switch` (мы узнаем о них позже). Если это кажется вам сложным, то вы не одиноки: из-за нюансов оператора `switch` он подвергся серьезной критике, поскольку является популярным источником ошибок программистов. Зачастую начинающим программистам не рекомендуют его использовать вообще. Но я полагаю, что оператор `switch` очень полезен в соответствующей ситуации: это хороший инструмент, который стоит иметь в своем арсенале, но, как и любым инструментом, им следует научиться владеть, использовать осторожно и там, где нужно.

Давайте начнем с очень простого примера оператора `switch`. Если у нашего суеверного моряка есть несколько предпочтительных чисел, мы можем использовать оператор `switch`, чтобы обработать их соответственно.

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 11:
        totalBet = 0;
        break;
    case 13:
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
}
```

Обратите внимание, что при ставках 11 и 13 осуществляется то же действие. Именно здесь нам и понадобиться свойство *аварийного перехода* управления. Помните, ранее упоминалось, что оператор `switch` продолжает выполнять операторы, пока не встретится оператор `break`. Используем это в своих интересах.

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 11:
    case 13:
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
}
```

До сих пор все довольно просто: понятно, что Томас ничего не будет ставить, если случайно вытащит 11 или 13 пенсов. Но что если число 13 выглядит намного более зловеще, чем 11, и требует не только воздержаться от ставки, но и отложить пенни на милостыню? Для реализации этого достаточно лишь небольшой переделки.

```
switch(totalBet) {
    case 7:
        totalBet = funds;
        break;
    case 13:
        funds = funds - 1; // подать 1 пенс милостыни!
    case 11:
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
}
```

Если `totalBet` составит 13, мы подаем пенни милостыни, но поскольку оператора `break` нет, выполняется переход к следующему разделу (11), в котором переменной `totalBet` присваивается значение 0. Этот код вполне допустим в JavaScript, а кроме того, он делает то, что мы и собирались сделать. Но у него действительно есть недостаток: он *выглядит* ошибочным (хотя и правильный). Вообразите, что ваш коллега увидел этот код и подумал "Так здесь же пропущен оператор `break`". Он добавит оператор `break`, и код больше не будет правильным. Многие вполне справедливо полагают, что аварийный переход несет больше неприятностей, чем он того стоит, но если вы решаете использовать его, то я рекомендую всегда включать комментарий, поясняющий, что вы сделали это намеренно.

Вы также можете определить частный случай, `default`, который будет использован, если никакой другой раздел не подходит. Обычно (но не обязательно), случай `default` располагают последним.

```
switch(totalBet) {
    case 7:
```

```
        totalBet = funds;
        break;
    case 13:
        funds = funds - 1; // подать 1 пенс милостыни!
    case 11:
        totalBet = 0;
        break;
    case 21:
        totalBet = 21;
        break;
    default:
        console.log("Здесь нет никаких суеверий!");
        break;
}
```

Оператор `break` здесь не нужен, поскольку никаких разделов за ним нет, однако добавление оператора `break` всегда является хорошей привычкой. Даже при использовании аварийного перехода нужно включать операторы `break`: вы всегда можете заменить оператор `break` комментарием, чтобы осуществить аварийный переход, однако отсутствие оператора `break`, когда он нужен, может стать ошибкой, которую очень трудно найти. Единственное исключение из этого эмпирического правила — используя оператор `switch` внутри функции (см. главу 6), вы можете заменять операторы `break` операторами `return` (поскольку они обеспечивают немедленный выход из функции).

```
function adjustBet(totalBet, funds) {
    switch(totalBet) {
        case 7:
            return funds;
        case 13:
            return 0;
        default:
            return totalBet;
    }
}
```

Как обычно, JavaScript не заботится о величине используемого отступа, поэтому оператор `break` (или `return`) весьма часто помещают в той же строке, чтобы сделать операторы `switch` более компактными.

```
switch(totalBet) {
    case 7: totalBet = funds; break;
    case 11: totalBet = 0; break;
    case 13: totalBet = 0; break;
    case 21: totalBet = 21; break;
}
```

Обратите внимание, что в этом примере мы решили повторить то же действие для 11 и 13 пенсов: отсутствие новой строки более ясно, когда в разделах есть одиночные операторы и нет аварийных переходов.

Операторы `switch` чрезвычайно удобны, когда необходимо выбрать один путь из многих на основании единого выражения. Как уже упоминалось, вы будете использовать их чаще, как только узнаете о динамической диспетчеризации в главе 9.

Цикл `for...in`

Цикл `for...in` предназначен для перебора *свойств объекта* по *ключам*. Его синтаксис таков.

```
for(переменная in объект)
    оператор
```

Вот пример его использования.

```
const player = { name: 'Thomas', rank: 'Midshipman', age: 25 };
for(let prop in player) {
    if(!player.hasOwnProperty(prop)) continue; // см. объяснение ниже
    console.log(prop + ': ' + player[prop]);
}
```

Не беспокойтесь, если сейчас не все понятно; вы узнаете больше об этом примере из главы 9. В частности, вызов `player.hasOwnProperty` не обязателен, но его отсутствие — популярный источник ошибок, рассматриваемый в главе 9. В настоящий момент достаточно знать, что это разновидность циклического оператора управления потоком.

Цикл `for...of`

Оператор `for...of` — нововведение ES6, обеспечивающее еще один способ циклического перебора элементов коллекции. Его синтаксис таков.

```
for(переменная of объект)
    оператор
```

Цикл `for...of` применим к массивам и в более общем случае к любым *итерируемым* (iterable) (см. главу 9) объектам. Вот пример его использования для перебора содержимого массива.

```
const hand = [randFace(), randFace(), randFace()];
for(let face of hand)
    console.log('You rolled...${face}!');
```

Цикл `for...of` — это прекрасный выбор, когда необходимо перебрать массив, он позволяет не заботиться об индексе каждого элемента. Если знать индексы необходимо, используйте обычный цикл `for`.

```
const hand = [randFace(), randFace(), randFace()];
for(let i=0; i<hand.length; i++)
    console.log('Roll ${i+1}: ${hand[i]}');
```

Популярные схемы управления потоком

Теперь, когда вы знаете основы конструкций управления потоком выполнения в JavaScript, давайте обратим внимание на некоторые из общепринятых схем, с которыми вы встретитесь.

Использование `continue` для сокращения содержимого условных выражений

Зачастую тело цикла необходимо продолжать выполнять только при определенных обстоятельствах (по существу, это комбинация цикла с условием), например так.

```
while(funds > 1 && funds < 100) {
    let totalBet = rand(1, funds);
    if(totalBet === 13) {
        console.log("Неудача! Пропустите этот раунд....");
    } else {
        // играть...
    }
}
```

Это пример *вложенности операторов управления потоком*; в теле цикла `while` есть директива `else` в которой выполняется действие, а в директиве `if` — всего лишь вызов `console.log`. Мы можем использовать оператор `continue`, чтобы упростить эту структуру.

```
while(funds > 1 && funds < 100) {
    let totalBet = rand(1, funds);
    if(totalBet === 13) {
        console.log("Неудача! Пропустите этот раунд....");
        continue;
    }
    // играть...
}
```

В этом простом примере преимущества не столь очевидны, но представьте, что тело цикла состоит не из 1 строки, а из 20; удалив эти строки из вложенного оператора управления потоком, мы сделаем код проще и понятнее.

Использование `break` или `return` во избежание ненужного вычисления

Если ваш цикл существует только для поиска чего-либо, а затем останавливается, то нет никакого смысла в выполнении всех этапов, если вы находите искомое на раннем этапе.

Предположим, например, что определение принадлежности числа к простым относительно дорого с точки зрения вычислений. Если вы ищете первый множитель в списке из тысяч чисел, прямолинейный подход мог бы быть таким.

```
let firstPrime = null;
for(let n of bigArrayOfNumbers) {
    if(isPrime(n) && firstPrime === null) firstPrime = n;
}
```

Если в `bigArrayOfNumbers` есть миллион чисел и только последнее является простым (что вам неизвестно), этот подход был бы прекрасен. Но что если простым является первое число? Или пятое, или пятидесятое? Вы проверили бы миллион чисел на принадлежность к простым, хотя могли бы остановиться в самом начале! Звучит удручающе. Мы можем использовать оператор `break`, как только найдем искомое.

```
let firstPrime = null;
for(let n of bigArrayOfNumbers) {
    if(isPrime(n)) {
        firstPrime = n;
        break;
    }
}
```

Если это цикл в функции, вместо `break` мы могли бы использовать оператор `return`.

Использование значения индекса после завершения цикла

Иногда важнейшим результатом цикла является значение индексной переменной после преждевременного завершении цикла оператором `break`. Мы можем использовать в своих интересах тот факт, что по завершении цикла `for` индексная переменная сохраняет свое значение. Если вы используете эту схему, имейте в виду случай, когда цикл завершается успешно и без `break`. Например, мы можем использовать эту схему для нахождения индекса первого простого числа в нашем массиве.

```
let i = 0;
for(; i < bigArrayOfNumbers.length; i++) {
    if(isPrime(bigArrayOfNumbers[i])) break;
}
if(i === bigArrayOfNumbers.length) console.log('Нет простых чисел!');
else console.log('Первое простое число находится в ${i} элементе массива.');
```

Использование убывающих индексов при изменении списков

Изменение списка в цикле по его элементам может быть очень сложным, поскольку при изменении списка могут измениться и условия выхода из цикла. В лучшем случае

результат окажется не тем, который вы ожидали получить, а в худшем вы можете получить бесконечный цикл. Обычно такую задачу решают, используя *убывающие* индексы, чтобы перебор осуществлялся с конца к началу. Таким образом, если вы добавите или удалите элементы из списка, то это не затронет условия завершения цикла.

Например, нам может понадобиться удалить все простые числа из массива `bigArrayOfNumbers`. Для этого мы используем метод `splice` объекта `Array`, позволяющий добавлять и удалять элементы из массива (см. главу 8). Следующее *не будет* работать, как ожидалось.

```
for(let i=0; i<bigArrayOfNumbers.length; i++) {
    if(isPrime(bigArrayOfNumbers[i])) bigArrayOfNumbers.splice(i, 1);
}
```

Поскольку индекс увеличивается, а мы удаляем элементы, есть вероятность, что мы перескочим через простое число (если оно окажется рядом). Мы можем решить эту проблему, используя убывающий индекс.

```
for(let i=bigArrayOfNumbers.length-1; i >= 0; i--) {
    if(isPrime(bigArrayOfNumbers[i])) bigArrayOfNumbers.splice(i, 1);
}
```

Обратите особое внимание на инициализацию и проверку условия выхода: мы начинаем со значения, которое *меньше* длины массива на единицу, поскольку индексы массивов отсчитываются от нуля. Кроме того, мы продолжаем цикл, пока `i` *больше или равно 0*; в противном случае этот цикл не обработает первый элемент в массиве (что стало бы проблемой, окажись первый элемент простым числом).

Заключение

Управление потоком действительно придал новый импульс нашим программам. Переменные и константы могут содержать всю интересную информацию, но операторы управления потоком позволяют нам сделать выбор на основании этих данных.

Блок-схемы — это удобное средство визуального описания потока. Зачастую имеет смысл описать задачу в виде высокоуровневой блок-схемы, прежде чем начинать писать код. Однако блок-схемы не очень компактны, и код — это более эффективное и (с практической точки зрения) более естественное средство выражения потока выполнения (кстати, было предпринято много попыток создать языки программирования, которые были бы исключительно визуальными, однако составить конкуренцию текстовым языкам они так и не смогли).

Выражения и операторы

Выражение (expression) — это специальный вид оператора, который *вычисляет значение*. Различие между оператором-выражением (который возвращает значение) и обычным оператором (который *не* возвращает значение) критически важно: его понимание дает инструмент, необходимый для объединения элементов языка.

Можно сравнить обычный оператор (т.е. *не* выражение) с *инструкцией*, а выражение — с *запросом чего-либо*. Предположим, что в ваш первый рабочий день на заводе заходит мастер и говорит: "Твоя задача — вворачивать болт А в фланец Б". Это оператор, а *не выражение*, так как мастер не требует показать собранную деталь; он просто проинструктировал вас, как выполнять сборку. Если бы мастер вместо этого сказал "Вверни болт А в фланец Б и покажи мне, что получилось", то это было бы эквивалентно *выражению*: мало того что вы следовали инструкции, вас еще попросили что-то возвратить. Вы же можете считать, что работа сделана в любом случае, поскольку собранная деталь существует независимо от того, осталась ли она на сборочном конвейере или передана мастеру для осмотра. В языке программирования это подобно оператору: он обычно *производит* нечто, но только выражения приводят к явной передаче произведенного результата.

Поскольку выражения возвращают значения, мы можем объединять их с другими выражениями, которые, в свою очередь, могут быть объединены с другими выражениями и т.д. Обычные операторы (не выражения), напротив, могли бы сделать нечто полезное, но они не могут быть объединены таким же образом.

Также вследствие того, что выражения возвращают значения, вы можете использовать их в операторах присваивания. Таким образом, вы можете присвоить результат выражения переменной, константе или свойству. Давайте рассмотрим распространенное *выражение*: операцию умножения. Вполне резонно, что умножение — это выражение: умножая два числа, вы получаете результат. Рассмотрим два очень простых оператора.

```
let x;
x = 3 * 5;
```

Первая строка — это *оператор* (statement) объявления; мы объявляем переменную x. Конечно, мы можем объединить эти две строки, но это затруднит данное обсуждение. Интересна также вторая строка: в ней фактически объединены два выражения. Первое *выражение* (expression), `3 * 5`, — это умножение, возвращающее результирующее значение `15`. Второе выражение — это присваивание значения `15` переменной x. Обратите внимание, что присваивание — это само по себе выражение, а выражение, как известно, возвращает значение. Так что же возвращает выражение присваивания? Как оказывается, выражения присваивания возвращают то значение, которое присваивают. Таким образом, не только переменной x будет присвоено значение `15`, но и *все выражение* возвратит значение `15`. Поскольку присваивание — это выражение, которое возвращает значение, мы могли бы, в свою очередь, присвоить его другой переменной. Считайте следующее (очень глупым) примером.

```
let x, y;
y = x = 3 * 5;
```

Теперь есть две переменные, x и y, и обе содержат значение `15`. Мы в состоянии сделать это потому, что и умножение, и присваивание являются выражениями. Когда в коде JavaScript встречается такое объединенное выражение, как это, оно разделяется и вычисляется по частям следующим образом.

```
let x, y;
y = x = 3 * 5; // исходный оператор
y = x = 15;    // вычисляется выражение умножения
y = 15;        // вычисляется первое присваивание; теперь x имеет
               // значение 15, а y все еще undefined
15;            // вычисляется второе присваивание; y теперь имеет
               // значение 15, результат – 15, который не используется
               // или присваивается чему-нибудь еще. Здесь от этого
               // заключительного значения просто отказываются
```

Вполне естественен вопрос — "Как JavaScript узнает, что выполнять выражения следует именно в таком порядке?" Вполне резонно полагать, что первым будет выполнена операция `y = x`, присваивающая переменной y значение `undefined`, а *затем* будет вычислен результат умножения и наконец — операция присваивания, оставляющая y значение `undefined`, а x — `15`. Порядок, в котором JavaScript обрабатывает выражения, зависит от *приоритета операторов* (operator precedence), и мы рассмотрим его в данной главе.

Большинство выражений, таких как умножение и присваивание, являются *операторами-выражениями*. Таким образом, выражение умножения состоит из *оператора умножения* (звездочка) и двух *операндов* (умножаемых значений, которые сами могут быть выражениями).

Операторами не являются всего два выражения: определение *идентификатора* (т.е. имен переменных и констант) и *литеральное выражение*. Это вполне очевидно, так как переменные, константы и литералы сами являются выражениями. Понимание этого позволяет вам увидеть, как с помощью выражений поддерживается

единообразие: если все, что приводит к значению, является выражением, то вполне резонно, что переменные, константы и литералы — это тоже выражения.

Операторы

Вы можете считать операторы "глаголами" для "существительных" выражений. Таким образом, *выражение* (expression) — это *нечто, приводящее к значению*; *оператор* (operator) — это нечто, что вы *делаете*, чтобы получить значение. Результат в обоих случаях — значение. Начнем наше обсуждение с арифметических операторов. Поскольку у большинства людей есть опыт выполнения арифметических действий, они интуитивно понятны.

Операторы обрабатывают один или несколько *операндов* (operand) и вычисляют результат. Например, в выражении `1 + 2` операнды — это `1` и `2`, а `+` — это оператор. Хотя технически правильный термин — *операнд*, их зачастую называют *аргументами* (argument).

Арифметические операторы

Арифметические операторы JavaScript приведены в табл. 5.1.

Таблица 5.1. Арифметические операторы

Оператор	Описание	Пример
`+`	Сложение (а также конкатенация строк)	`3 + 2 // 5`
`-`	Вычитание	`3 - 2 // 1`
`/`	Деление	`3/2 // 1.5`
`*`	Умножение	`3*2 // 6`
`%`	Остаток	`3%2 // 1`
`-`	Унарное вычитание	`-x // Отрицательное x; если x равна 5, то -x равно -5`
`+`	Унарная сумма	`+x // Если x не будет числом, то произойдетпопытка преобразования`
`++`	Префиксный инкремент	`++x // Инкремент x на единицу и возвращение нового значения`
`++`	Постфиксный инкремент	`x++ // Инкремент x на единицу и возвращение старого значения x (т.е. до инкремента)`
`--`	Префиксный декремент	`--x // Декремент x на единицу и возвращение нового значения`
`--`	Постфиксный декремент	`x-- // Декремент x на единицу и возвращение старого значения x (т.е. до декремента)`

Помните, что все числа в JavaScript имеют двойную точность, а значит, если вы выполняете арифметическую операцию с целыми числами (например, `3/2`), то результат имеет вид десятичного числа (`1.5`).

Для вычитания и унарного вычитания используется одинаковый символ (знак "минус"), так как же JavaScript узнает, что им обозначено? Ответ на этот вопрос сложен и здесь не рассматривается. Достаточно знать, что унарное вычитание вычисляется перед обычным.

```
const x = 5;
const y = 3 - -x;   // y равно 8
```

То же относится к унарной сумме. Унарная сумма используется довольно редко. Обычно с ее помощью строка преобразуется в число или она используется для выравнивания значений при инвертировании знака некоторых из них.

```
const s = "5";
const y = 3 + +s; // y равно 8; без унарного плюса это
                  // был бы результат конкатенации строк: "35"

// использование ненужных унарных плюсов для выравнивания кода
const x1 = 0, x2 = 3, x3 = -1.5, x4 = -6.33;
const p1 = -x1*1;
const p2 = +x2*2;
const p3 = +x3*3;
const p3 = -x4*4;
```

Обратите внимание, что в этих примерах я специально использовал переменные с операторами унарного вычитания и суммы. Если вы используете их с числовыми литералами, то знак "минус" фактически становится частью числового литерала, а потому технически он не оператор.

Оператор остатка возвращает остаток после деления. Если у вас есть выражение `x % y`, то результатом будет остаток от делении *делимого* (`x`) на *делитель* (`y`). Например, `10 % 3` даст `1` (3 входит в 10 три раза и остается 1). Обратите внимание, что при отрицательных числах результат получит знак делимого, а не делителя, что не позволяет ему быть истинным оператором модуля. В то время как оператор остатка обычно используется только с целочисленными операндами, в JavaScript он применим и к дробным операндам. Например, `10 % 3.6` даст `3` (`3.6` входит в 10 дважды, остается 2.8).

Оператор инкремента (++) фактически является оператором суммы и оператором присваивания одновременно. Аналогично оператор декремента (--) является оператором вычитания и присваивания. Это очень полезные сокращения, но использовать их следует осторожно: если применить один из них глубоко в выражении, то он может стать причиной трудно обнаруживаемого "побочного эффекта" (изменений в переменной). Понимание различий между *префиксными* (prefix) и *постфиксными*

(postfix) операторами также важно. Префиксная версия изменяет значение переменной, а затем возвращает *новое* значение; постфиксный оператор изменяет значение переменной, а затем возвращает *первоначальное* значение. Посмотрим, сможете ли вы предсказать результаты вычисления следующих выражений (подсказка: операторы инкремента и декремента вычисляются перед сложением, а вычисления в этом примере осуществляются слева направо).

```
let x = 2;
const r1 = x++ + x++;
const r2 = ++x + ++x;
const r3 = x++ + ++x;
const r4 = ++x + x++;
let y = 10;
const r5 = y-- + y--;
const r6 = --y + --y;
const r7 = y-- + --y;
const r8 = --y + y--;
```

Давайте запустим этот пример на консоли JavaScript и выясним, сможете ли вы предсказать значения констант `r1` – `r8`, а также значения `x` и `y` на каждом этапе. Если у вас возникли проблемы с этим упражнением, попробуйте записать задачу на листе бумаги и расставить круглые скобки согласно порядку выполнения каждой операции, например так.

```
let x = 2;
const r1 = x++ + x++;
//       ((x++) + (x++))
//          ( 2 + (x++))    вычисляется слева направо;
//                          теперь x содержит значение 3
//          ( 2 + 3 )       теперь x содержит значение 4
//              5           результат 5; x содержит значение 4
const r2 = ++x + ++x;
//       ((++x) + (++x))
//          ( 5 + (++x))    вычисляется слева направо;
//                          теперь x содержит значение 5
//          ( 5 + 6 )       теперь x содержит значение 6
//              11          результат 11; x содержит значение 6
const r3 = x++ + ++x;
//       ((x++) + (++x))
//          ( 6 + (++x))    вычисляется слева направо;
//                          теперь x содержит значение 7
//          ( 6 + 8 )       теперь x содержит значение 8
//              14          результат 14; x содержит значение 8
//
// ... и так далее
```

Приоритет операторов

Вторая по важности концепция, необходимая для осознания того, как каждое выражение возвращает значение, приоритет операторов, — это следующий жизненно важный этап для понимания работы программ JavaScript.

Теперь, рассмотрев арифметические операторы, давайте приостановим наше обсуждение набора операторов JavaScript и обсудим их приоритет. Если вы учились в школе, то должны быть уже знакомы с понятием приоритета операторов, даже если еще и не подозреваете об этом.

Помните, когда-то в младших классах вы решали такие задачи (заранее прошу прощения у тех, кого раздражала арифметика).

```
8 ÷ 2 + 3 × (4 × 2 - 1)
```

Если ваш ответ — 25, вы правильно применили приоритет операторов. Вы знали, что нужно начинать с круглых скобок, затем переходить к умножению и делению, а завершать сложением и вычитанием.

Для принятия решения о том, как вычислить любое выражение (а не только арифметическое), JavaScript использует подобный свод правил. Вы будете рады узнать, что в арифметических выражениях в JavaScript используется тот же порядок операций, который вы изучали в младших классах школы, легко запоминаемый с помощью мнемоники "PEMDAS" или "Please Excuse My Dear Aunt Sally" (Пожалуйста, извините мою дорогую тетю Салли).

Кроме арифметических в JavaScript есть еще много операторов, поэтому плохая новость: запомнить вам придется намного больший порядок. Хорошая новость — здесь, как и в математике, круглые скобки превосходят все: если вы не уверены в порядке выполнения операций определенного выражения, всегда можете поместить в круглые скобки операции, которые нужно выполнять сначала.

В настоящее время в JavaScript есть 56 операторов, сгруппированных в 19 *уровней приоритета* (precedence level). Операторы с более высоким приоритетом выполняются перед операторами с более низким приоритетом. Хотя с годами я постепенно запомнил эту таблицу (не предпринимая для этого сознательных усилий), я все еще иногда обращаюсь к ней, чтобы обновить ее в памяти или увидеть, в какие уровни приоритета вписываются новые средства языка. Таблица приоритета операторов приведена в приложении Б.

Операторы с одинаковым приоритетом вычисляются либо *справа налево*, либо *слева направо*. Например, умножение и деление имеют один и тот же приоритет (14) и вычисляются слева направо, а операторы присваивания (уровень приоритета — 3) вычисляются справа налево. Вооружившись этим знанием, мы можем определить порядок выполнения операций в данном примере.

```
let x = 3, y;
x += y = 6*5/2;
// расставим круглые скобки вокруг следующих операций согласно
// приоритету:
//
// умножение и деление (уровень приоритета — 14, слева направо):
//    x += y = (6*5)/2
//    x += y = (30/2)
//    x += y = 15
// присваивание (уровень приоритета — 3, справа налево):
//    x += (y = 15)
//    x += 15          (теперь y содержит значение 15)
//    18               (теперь x содержит значение 18)
```

Поначалу понять приоритет операторов может быть сложно, но потом это быстро становится второй натурой.

Операторы сравнения

Операторы сравнения, как и следует из их названия, используются для сравнения двух значений. Есть три типа оператора сравнения: строгое равенство, абстрактное (или свободное) равенство и сравнение. (Мы не рассматриваем неравенства как отдельный тип: неравенство — это просто "не равенство", даже при том что у него есть собственный оператор для удобства.)

Новичкам труднее всего понять различие между *строгим равенством* (strict equality) и *абстрактным равенством* (abstract equality). Начнем со строгого равенства, поскольку я рекомендую всегда предпочитать именно его. Два значения считаются строго равными, если они ссылаются на один и тот же объект или имеют один и тот же тип и одно и то же значение (у базовых типов). Преимущество строгого равенства в том, что его правила очень просты и понятны, что делает его менее склонным к ошибкам и недоразумениям. Чтобы определить, строго ли равны значения, используйте оператор `===` или его противоположность — оператор строгого неравенства (`!==`). Прежде чем мы рассмотрим несколько примеров, давайте обсудим оператор абстрактного равенства.

Два значения считаются абстрактно равными, если они ссылаются на один и тот же объект (и то хорошо) или *если они могут быть приведены к одному и тому же значению*. Именно эта вторая часть вызывает так много неприятностей и недопонимания. Иногда это свойство полезно. Например, если вы захотите узнать, равны ли число `33` и строка `"33"`, то оператор абстрактного равенства скажет, что равны, но оператор строгого равенства скажет, что не равны (поскольку они имеют разные типы). Хотя, на первый взгляд, абстрактное равенство может показаться более удобным, вы вместе с этим удобством можете получить больше непредвиденного поведения. Поэтому я рекомендую заранее преобразовывать строки в числа; таким

образом, вы сможете сравнивать их, используя оператор строгого равенства. Оператор абстрактного равенства — это ==, а абстрактного неравенства — !=. Если вы хотите иметь подробную информацию о проблемах с оператором абстрактного равенства, я рекомендую книгу Дугласа Крокфорда *JavaScript: The Good Parts* (издательство O'Reilly).

Большинство проблем поведения операторов абстрактного равенства связано со значениями `null`, `undefined`, пустой строкой и числом 0. По большей части, если вы сравниваете значения, о которых известно, что они не могут быть одним из этих значений, использовать оператор абстрактного равенства совершенно безопасно. Однако не стоит недооценивать силу механической привычки. Если вы решаете, как я рекомендую, всегда использовать оператор строгого равенства, то вы можете *никогда не заботиться об этом*. Вам не придется прерывать ход своей мысли, чтобы задаться вопросом, будет ли безопасно или выгодно использовать оператор абстрактного равенства; вы просто используете оператор строгого равенства и двигаетесь дальше. Если впоследствии окажется, что оператор строгого равенства не дает желаемого результата, вы можете сделать соответствующее преобразование типов вместо того, чтобы переходить на проблематичный оператор абстрактного равенства. Программирование — и так достаточно сложное занятие, поэтому сделайте себе одолжение и избегайте проблематичного оператора абстрактного равенства.

Вот несколько примеров использования операторов строгого и абстрактного равенства. Обратите внимание: даже при том, что объекты `a` и `b` содержат одну и ту же информацию, *это разные объекты*, и они ни строго, ни абстрактно не равны.

```
const n = 5;
const s = "5";
n === s;           // false -- разные типы
n !== s;           // true
n === Number(s);   // true -- "5" преобразуется в число 5
n !== Number(s);   // false
n == s;            // true;  не рекомендуется
n != s;            // false; не рекомендуется
const a = { name: "an object" };
const b = { name: "an object" };
a === b;           // false -- разные объекты
a !== b;           // true
a == b;            // false; не рекомендуется
a != b;            // true;  не рекомендуется
```

Операторы сравнения сравнивают значения один *относительно* другого и имеют смысл только для естественно упорядоченных типов данных, таких как строки ("a" предшествует "b") и числа (0 меньше 1). Операторы сравнения — меньше (<), меньше или равно (<=), больше (>) и больше или равно (>=).

```
3 > 5;    // false
3 >= 5;   // false
3 < 5;    // true
3 <= 5;   // true
5 > 5;    // false
5 >= 5;   // true
5 < 5;    // false
5 <= 5;   // true
```

Сравнение чисел

При сравнении идентификаторов или чисел на равенство следует соблюдать особую осторожность.

В первую очередь, обратите внимание на то, что специальное числовое значение NaN не равно ничему, включая само себя (т.е. и NaN === NaN, и NaN == NaN дают false). Если вы хотите проверить, не является ли число NaN, используйте встроенную функцию isNaN: вызов isNaN(x) возвратит true, если x является NaN, и false — в противном случае.

Напомню, что все числа в JavaScript — двойной точности, а поскольку для этого типа характерны округления (если это необходимо), вы можете столкнуться с неприятными неожиданностями при их сравнении.

Сравнивая целые числа (от Number.MIN_SAFE_INTEGER до Number.MAX_SAFE_INTEGER включительно), вы можете безопасно использовать тождество для проверки равенства. При использовании дробных чисел лучше прибегнуть к оператору сравнения, чтобы убедиться, что проверяемое вами число "достаточно близко" к исходному числу. Что значит "достаточно близко"? Это зависит от вашего приложения. В JavaScript доступна специальная числовая константа, Number.EPSILON. Это очень маленькое значение (2.22e-16), представляющее разницу, необходимую для двух чисел, чтобы они считались различными. Рассмотрим пример.

```
let n = 0;
while(true) {
    n += 0.1;
    if(n === 0.3) break;
}
console.log('Stopped at ${n}');
```

Попытавшись запустить эту программу, вы будете неприятно удивлены: вместо того чтобы остановиться на значении 0.3, этот цикл пропустит его и продолжит

выполняться бесконечно. Это связано с тем, что `0.1` — это число, которое не может быть точно представлено в виде значения двойной точности, поскольку оно находится между двумя представлениями двоичной дроби. Поэтому на третьей итерации этого цикла `n` будет иметь значение `0.30000000000000004`, проверка даст `false` и единственный шанс прервать цикл будет упущен.

Этот цикл можно переписать так, чтобы использовать число `Number.EPSILON` и оператор сравнения, чтобы осуществить "более мягкое" сравнение и успешно остановить цикл.

```
let n = 0;
while(true) {
    n += 0.1;
    if(Math.abs(n - 0.3) < Number.EPSILON) break;
}
console.log('Stopped at ${n}');
```

Обратите внимание, что мы вычитаем целевое число (`0.3`) из проверяемого числа (`n`) и получаем его абсолютное значение (используя функцию `Math.abs`, рассмотренную в главе 16), которое затем и сравниваем с пороговым значением. В данном случае мы могли бы просто проверить, не больше ли `n` значения `0.3`. Однако мы выполнили несложные вычисления, чтобы продемонстрировать вам способ определения, достаточно ли два числа двойной точности близки, чтобы их можно считать равными.

Конкатенация строк

В JavaScript оператор + используется и для сложения чисел, и для конкатенации строк (что весьма распространено; наиболее известными примерами языков, *не использующих* оператор + для конкатенации строк, являются Perl и PHP).

Что именно имеется в виду, сложение чисел или конкатенации строк, JavaScript определяет по типами операндов. И сложение, и конкатенация обрабатываются слева направо. JavaScript исследует каждую пару операндов слева направо, и если любой из операндов является строкой, то осуществляется конкатенация строк. Если оба значения являются числовыми, то происходит сложение. Рассмотрим следующие две строки кода.

```
3 + 5 + "8"     // результат - строка "88"
"3" + 5 + 8     // результат - строка "358"
```

В первом случае JavaScript сначала вычисляет сумму `(3 + 5)`, а затем осуществляет конкатенацию строк `(8 + "8")`. Во втором случае `("3" + 5)` вычисляется как конкатенация, затем `("35" + 8)` — тоже как конкатенация.

Логические операторы

Принимая во внимание, что арифметические операторы, с которыми мы все знакомы, работают с числами, способными принимать бесконечные количества значений (или по крайней мере очень большое количество значений, поскольку память компьютеров конечна), логические операторы обрабатывают только логические значения, способные принимать только одно из двух значений: истина или ложь.

В математике (и во многих языках программирования) логические операторы работают только с логическими значениями и возвращают только логические значения. JavaScript позволяет работать со значениями, не являющимися логическими, и, что еще удивительнее, способными возвращать значения, не являющиеся логическими. Я не хочу сказать, что реализация логических операторов в JavaScript является в чем-то неправильной или не строгой: используя только логические значения, вы будете получать результаты, являющиеся только логическими значениями.

Прежде чем мы обсудим сами операторы, необходимо ознакомиться с механизмом сопоставления не логических значений с логическими значениями в JavaScript.

Истинные и ложные значения

Во многих языках есть концепция значений "истинности" и "ложности"; в C, например, даже нет логического типа: число 0 — это ложь, а все другие числовые значения — истина. В JavaScript происходит нечто подобное, но задействуются все типы данных, фактически позволяя различить любое значение как истину и ложь. JavaScript полагает следующие значения ложью (`false`).

- `undefined`
- `null`
- `false`
- `0`
- `NaN`
- `''` (пустая строка)

Все остальное — истина (`true`). Поскольку истиной является великое множество значений, я не буду перечислять их здесь все, я укажу лишь те, о которых следует знать.

- Любой объект (включая тот, метод `valueOf()` которого возвращает `false`).
- Любой массив (даже пустой).
- Строки, содержащие только отступ (например, `" "`).
- Строка `"false"`.

Некоторых смущает тот факт, что строка `"false"` (ложь) является истиной (`true`), но по большей части это имеет смысл и легко запоминается. Еще один известный факт, что пустой массив — это `true`. Если вы хотите, чтобы массив `arr` давал `false`, когда он пуст, то используйте `arr.length` (что даст `0`, если массив пуст, а следовательно, `false`).

Операторы AND, OR и NOT

JavaScript поддерживает три логических оператора: AND (`&&`), OR (`||`) и NOT (`!`). Если у вас есть математическая подготовка, вы знаете, что AND (И) — это конъюнкция, OR (ИЛИ) — дизъюнкция и NOT (НЕ) — инверсия.

В отличие от чисел, количество возможных значений которых бесконечно, логические переменные могут иметь только два возможных значения, поэтому эти операции зачастую описываются *таблицами истинности*, в которых полностью описано их поведение (см. табл. 5.2–5.4).

Таблица 5.2. Таблица истинности для оператора AND (`&&`)

X	Y	X && Y
`false`	`false`	`false`
`false`	`true`	`false`
`true`	`false`	`false`
`true`	`true`	`true`

Таблица 5.3. Таблица истинности для оператора OR (`||`)

X	Y	X || Y
`false`	`false`	`false`
`false`	`true`	`true`
`true`	`false`	`true`
`true`	`true`	`true`

Таблица 5.4. Таблица истинности для оператора NOT (`!`)

X	!X
`false`	`true`
`true`	`false`

Как можно заметить из этих таблиц, результатом работы оператора AND будет `true`, только если оба из его операндов — `true`; результатом оператора OR будет `false`, только если оба его операнда — `false`. Оператор NOT прост, он получает один операнд и возвращает обратное значение.

Оператор OR иногда называют включающим OR ("inclusive OR"), поскольку, если оба его операнда `true`, результат — `true`. Есть также исключающее OR ("exclusive OR") или XOR, возвращающий `false`, если оба операнда — `true`. В JavaScript нет логического оператора XOR, но есть побитовый XOR, обсуждаемый далее.

Если необходим результат логического исключающего OR (XOR) двух переменных, `x` и `y`, вы можете воспользоваться эквивалентным выражением `(x || y) && x !== y`.

Вычисление по сокращенной схеме

Если вы просмотрите таблицу истинности для оператора AND (см. табл. 5.2), то обратите внимание, что возможно сокращение: если `x` — ложь, рассматривать значение `y` уже не нужно. Точно так же, если вы вычисляете `x || y` и `x` — истина, то вычислять `y` уже не нужно. Именно это и делает JavaScript в ходе *вычисления по сокращенной схеме* (short-circuit evaluation).

Почему так важно знать о вычислении по сокращенной схеме? Потому что, если у второго операнда есть *побочные эффекты* (side effect), при сокращении вычисления их не будет. Очень часто термин "побочный эффект" применяется для описания чего-то плохого, но в программировании это не всегда так: если побочный эффект вызван намеренно, то это неплохо.

В выражении побочные эффекты могут явиться результатом инкремента, декремента, присваивания и вызова функции. Мы уже рассматривали операторы инкремента и декремента, поэтому давайте приведем пример.

```
const skipIt = true;
let x = 0;
const result = skipIt || x++;
```

В третьей строке этого примера получается очевидный результат, сохраняемый в переменной `result`. Этим значением будет `true`, поскольку первый операнд (`skipIt`) — это `true`. Интересно, тем не менее, то, что из-за вычисления по сокращенной схеме выражение инкремента не вычисляется, оставляя `x` значение `0`. Если изменить значение `skipIt` на `false`, то обе части выражения будут вычислены и инкремент выполнится: здесь инкремент — это побочный эффект. То же самое относится к оператору И.

```
const doIt = false;
let x = 0;
const result = doIt && x++;
```

Здесь JavaScript снова не станет вычислять второй операнд, который содержит инкремент, поскольку первым операндом AND будет `false`. Таким образом, `result` содержит `false`, а `x` остался без инкремента. Что будет, измени вы `doIt` на `true`? JavaScript должен вычислить оба операнда; таким образом, инкремент произойдет и `result` будет `0`. Не ожидали? Почему константа `result` содержит значение `0`, а не `false`? Ответ на этот вопрос плавно переводит нас к следующей теме.

Логические операторы с не логическими операндами

Если вы используете логические операнды, логические операторы всегда возвращают логические значения. В противном случае возвращаемое значение будет *зависеть от типов операндов*, как показано в табл. 5.5 и 5.6.

Таблица 5.5. Таблица истинности для оператора AND (&&) с не логическими операндами

X	Y	X && Y
ложь	ложь	X (ложь)
ложь	истина	X (ложь)
истина	ложь	Y (ложь)
истина	истина	Y (истина)

Таблица 5.6. Таблица истинности для оператора OR (||) с не логическими операндами

X	Y	X || Y
ложь	ложь	Y (ложь)
ложь	истина	Y (истина)
истина	ложь	Y (истина)
истина	истина	Y (истина)

Обратите внимание, что если вы преобразуете результат в логическое значение, то он будет правильным согласно определениям логических операторов AND и OR. Подобное поведение логических операторов допускает определенные весьма удобные сокращения. Вот одно из достаточно наглядных.

```
const options = suppliedOptions || { name: "Default" }
```

Помните, что объекты (даже если они пусты) всегда считаются истиной. Таким образом, если `suppliedOptions` — это объект, то `options` ссылается на `suppliedOptions`. Если никаких параметров не предоставлено, то `suppliedOptions` будет `null` или `undefined`, а переменная `options` получит некое стандартное значение.

В случае оператора NOT нет никакого резонного способа возвратить нечто, кроме логического значения, поэтому оператор (`!`) всегда возвращает логическое значение для операнда любого типа. Если операнд истинный, он возвращает `false`, в противном случае — `true`.

Условный оператор

Условный оператор — это единственный *тройственный оператор* (ternary) JavaScript: он получает три операнда (все остальные получают один или два

операнда). Условный оператор — выражение, эквивалентное оператору `if...else`. Вот пример условного оператора.

```
const doIt = false;
const result = doIt ? "Сделай это!" : "Расслабься!";
```

Если первый операнд (тот который находится перед вопросительным знаком, в данном случае — `doIt`) истинный, выражение возвращает второй операнд (расположенный между вопросительным знаком и двоеточием), а если нет, то третий (находящийся после двоеточия). Много начинающих программистов видят это как более сложную версию оператора `if...else`, однако фактически это не оператор, а выражение, обладающее очень полезным свойством: оно может быть объединено с другими выражениями (такими, как присваивание переменной `result` в последнем примере).

Оператор "запятая"

Оператор "запятая" обеспечивает простой способ объединения выражений: просто вычисляет два выражения и возвращает результат второго. Это очень удобно, если вы хотите выполнить несколько выражений, но единственное значение, о котором вы заботитесь, является результатом заключительного выражения. Вот простой пример.

```
let x = 0, y = 10, z;
z = (x++, y++);
```

В этом примере происходит инкремент обеих переменных, `x` и `y`, а переменная `z` получает значение `10` (возвращаемое `y++`). Обратите внимание, что у оператора запятой самый низкий приоритет среди всех операторов, поэтому мы и заключили его здесь в круглые скобки. Без них переменная `z` получила бы значение `0` (значение `x++`), а *затем* произошел бы инкремент переменной `y`. Обычно он применяется для объединения выражений в цикле `for` (см. главу 4) или для объединения нескольких операций перед выходом из функции (см. главу 6).

Оператор группировки

Как уже упоминалось, оператор группировки (круглые скобки) не делает ничего, кроме изменения (и разъяснения) приоритета операторов. Таким образом, оператор группировки — "самый безопасный" оператор, который влияет только на порядок выполнения операций.

Побитовые операторы

Побитовые операторы позволяют выполнять операции с *отдельными* битами числа. Если у вас нет никакого опыта работы с низкоуровневым языком

программирования, таким как ассемблер или C[1], или нет никакого представления о том, как компьютер внутренне хранит числа, вам, возможно, захочется изучить сначала эти темы (вы также можете пропустить или только просмотреть этот раздел, поскольку лишь в некоторых приложениях требуются побитовые операторы). Побитовые операторы обрабатывают свои операнды как 32-разрядные целые числа со знаком, представленные в *двоичном дополнительном коде*. Поскольку все числа в JavaScript — с двойной точностью, JavaScript преобразует их в 32-разрядные целые числа, прежде чем выполнять побитовые операторы, а затем преобразует их обратно в числа двойной точности, прежде чем возвратить результат.

Побитовые операторы, подобно логическим операторам, выполняют логические операции (И, ИЛИ, НЕ, XOR), но с каждым отдельным битом целого числа. В табл. 5.7 показано, что среди побитовых операторов есть также операторы *сдвига*, позволяющие переместить все биты числа влево или вправо на нужно количество разрядов.

Таблица 5.7. Побитовые операторы

Оператор	Описание	Пример
`&`	Побитовое AND	`0b1010 & 0b1100 // результат: 0b1000`
`|`	Побитовое OR	`0b1010 | 0b1100 // результат: 0b1110`
`^`	Побитовое XOR	`0b1010 ^ 0b1100 // результат: 0b0110`
`~`	Побитовое NOT	`~0b1010 // результат: 0b0101`
`<<`	Сдвиг влево	`0b1010 << 1 // результат: 0b10100` `0b1010 << 2 // результат: 0b101000`
`>>`	Сдвиг вправо	(См. ниже)
`>>>`	Сдвиг вправо с заполнением нулями	(См. ниже)

Обратите внимание, что сдвиг влево фактически умножает число на два, а сдвиг вправо фактически делит его на два и округляет в меньшую сторону.

В двоичном дополнительном коде крайний слева бит обозначает знак числа. Для отрицательных чисел он содержит 1, а для положительных — 0, следовательно, есть два способа выполнить сдвиг вправо. Давайте возьмем, например, число –22. Если мы хотим получить его двоичное представление, начнем с положительного числа 22, представим его в *обратном коде* (т.е. инвертируем все его биты), а затем добавим к полученному результату единицу, чтобы получить *двоичный дополнительный код*.

```
let n = 22  // 32-разрядное двоичное число:
            //                          00000000000000000000000000010110
n >> 1      //                          00000000000000000000000000001011
n >>> 1     //                          00000000000000000000000000001011
n = ~n      // обратный код числа: 11111111111111111111111111101001
```

[1] Язык C многие современные эксперты в программировании относят к низкоуровневым, хотя, строго говоря, это не так. — *Примеч. ред.*

```
n++          // двоичный доп. код:  11111111111111111111111111101010
n >> 1       //                     11111111111111111111111111110101
n >>> 1      //                     01111111111111111111111111110101
```

Если только вы не взаимодействуете с аппаратными средствами или не собираетесь лучше узнавать, как числа представляются на компьютере, вам вряд ли пригодятся побитовые операторы (зачастую этот процесс игриво называют "жонглированием битами"). Один из не связанных с аппаратными средствами случаев их применения — это использование битов для эффективного хранения "флагов" (логических значений).

Рассмотрим, например, установку прав доступа к файлу в системах Unix: на чтение (read), на запись (write) и на выполнение (execute). Пользователь может задать любую комбинацию этих трех параметров, что делает их идеальными для флагов. Поскольку имеется три флага, необходимо три бита для хранения информации.

```
const FLAG_READ 1    // 0b001
const FLAG_WRITE 2   // 0b010
const FLAG_EXECUTE 4 // 0b100
```

Используя побитовые операторы мы можем объединять, переключать и возвращать значения отдельных флагов, которые хранятся в едином числовом значении.

```
let p = FLAG_READ | FLAG_WRITE;     // 0b011
let hasWrite = p & FLAG_WRITE;      // 0b010 - истина
let hasExecute = p & FLAG_EXECUTE; // 0b000 - ложь
p = p ^ FLAG_WRITE;                 // 0b001 -- переключение флага write
                                    // (теперь он сброшен)
p = p ^ FLAG_WRITE;                 // 0b011 -- переключение флага write
                                    // (теперь он установлен)

// мы можем определить значение даже нескольких флагов
// с помощью одного выражения:
const hasReadAndExecute = p & (FLAG_READ | FLAG_EXECUTE);
```

Обратите внимание, что для `hasReadAndExecute` мы должны были использовать оператор группировки; у оператора AND приоритет более высокий, чем у OR, а нам нужно было выполнить OR до AND.

Оператор `typeof`

Оператор `typeof` возвращает строку, представляющую тип его операнда. К сожалению, этот оператор не обеспечивает точного соответствия семи типам данных JavaScript (`undefined`, `null`, логический, число, строка, символ и объект), что стало причиной бесконечной критики и затруднений.

У оператора `typeof` есть одна причуда, которую обычно называют ошибкой: `typeof null` возвращает `"object"`. Конечно, `null` — никакой не объект (это

базовый тип). Причины имеют исторический характер и не особенно интересны, уже неоднократно предлагалось исправить это, но слишком много существующего кода уже основано на таком поведении. В результате теперь это увековечено в спецификации языка.

Оператор `typeof` нередко критикуют также за то, что он не способен отличить объекты, не являющиеся массивами, от массивов. Он корректно идентифицирует функции (также являющиеся объектами специальных типов), но результатом `typeof []` является `"object"`.

Возможные возвращаемые значения оператора `typeof` приведены в табл. 5.8.

Таблица 5.8. Возвращаемые значения оператора `typeof`

Выражение	Возвращаемое значение	Примечания
`Typeof undefined`	`"undefined"`	
`Typeof null`	`"object"`	Прискорбно, но факт
`Typeof {}`	`"object"`	
`Typeof true`	`"boolean"`	
`Typeof 1`	`"number"`	
`Typeof ""`	`"string"`	
`Typeof Symbol()`	`"symbol"`	Нововведение ES6
`Typeof function() {}`	`"function"`	

Поскольку `typeof` — оператор, круглые скобки не обязательны. Таким образом, если у вас есть переменная `x`, можете использовать синтаксис `typeof x` вместо `typeof (x)`. Последнее — вполне допустимый синтаксис — круглые скобки лишь создают ненужную группу выражений.

Оператор `void`

У оператора `void` есть только одна задача: вычислить свой операнд, а затем возвратить `undefined`. Звучит не очень привлекательно? Да, согласен. Это применяется для оценки выражения, в котором необходимо возвращаемое значение `undefined`, но я никогда не встречался с такой ситуаций в реальной жизни. Единственная причина, по которой я включил его в эту книгу, в том, что вы будете иногда встречать его использование в URL дескриптора HTML `<a>`, чтобы помешать браузеру перейти на новую страницу.

```
<a href="javascript:void 0">Ничего не делаем! </a>
```

Это не рекомендуемый подход, но время от времени он встречается.

Операторы присваивания

Оператор присваивания прост: он назначает значение переменной. То, что находится слева от знака равенства (иногда называемое *l-значением* (lvalue)), *должно*

быть переменной, свойством или элементом массива. Таким образом, это нечто, способное содержать значение (присваивание значения константе технически является частью объявления, а не оператором присваивания).

Ранее в этой главе упоминалось, что операция присваивания сама по себе является выражением, а следовательно, она возвращает значение (а именно — присваиваемое значение). Это позволяет объединять операторы присваивания в цепочки и выполнять присваивания в пределах других выражений.

```
let v, v0;
v = v0 = 9.8; // сцепленное присваивание; сначала v0 получает
              // значение 9.8, а затем v получает значение 9.8

const nums = [ 3, 5, 15, 7, 5 ];
let n, i=0;
// обратите внимание на оператор присваивания в условии цикла while; n получает
// значение nums[i] и все выражение также получает это значение,
// обеспечивая числовое сравнение:
while((n = nums[i]) < 10, i++ < nums.length) {
    console.log('Число меньше 10: ${n}.');
}
console.log('Найдено число больше 10: ${n}.');
console.log('${nums.length} всего чисел.');
```

Обратите внимание на то, что во втором примере используется оператор группировки, поскольку приоритет оператора присваивания ниже приоритета оператора сравнения.

Кроме обычных операторов присваивания, есть *составные операторы присваивания*, выполняющие операцию и присваивание за один этап. Подобно обычным операторам присваивания, эти операторы вычисляют результирующее значение присваивания. Составные операторы присваивания приведены в табл. 5.9.

Таблица 5.9. Составные операторы присваивания

Оператор	Эквивалент
`x += y`	`x = x + y`
`x -= y`	`x = x - y`
`x *= y`	`x = x * y`
`x /= y`	`x = x / y`
`x %= y`	`x = x % y`
`x <<= y`	`x = x << y`
`x >>= y`	`x = x >> y`
`x >>>= y`	`x = x >>> y`
`x &= y`	`x = x & y`
`x |= y`	`x = x | y`
`x ^= y`	`x = x ^ y`

Деструктурирующее присваивание

Нововведением ES6 является *деструктурирующее присваивание* (destructuring assignment), позволяющее взять объект или массив и "деструктурировать" его на индивидуальные переменные. Начнем с деструктуризации объекта.

```
// обычный объект
const obj = { b: 2, c: 3, d: 4 };

// деструктурирующее присваивание объекта
const {a, b, c} = obj;
a;              // undefined: в obj нет свойства "a"
b;              // 2
c;              // 3
d;              // ошибка ссылки: "d" не определено
```

При деструктуризации объекта имена переменных должны соответствовать именам свойства объекта (при деструктуризации массива можно использовать любые корректные имена переменных). В этом примере переменная `a` не соответствует свойству в объекте, поэтому она получила значение `undefined`. Кроме того, поскольку в объявлении мы не определили переменную `d`, ей ничего не было присвоено.

В этом примере мы осуществляем объявление и присваивание в том же операторе. Деструктуризация объекта может быть осуществлена с помощью только одного оператора присваивания, но его следует заключить в круглые скобки; в противном случае JavaScript интерпретирует его левую сторону как блок.

```
const obj = { b: 2, c: 3, d: 4 };
let a, b, c;

// это приведет к ошибке:
{a, b, c} = obj;

// это сработает:
({a, b, c} = obj);
```

При деструктуризации массива вы можете назначить элементам массива любые имена по своему усмотрению.

```
// обычный массив
const arr = [1, 2, 3];

// деструктурирующее присваивание массива
let [x, y] = arr;
x;              // 1
y;              // 2
z;              // ошибка: z не была определена
```

В этом примере `x` присваивается значение первого элемента массива, а `y` — второго; все остальные элементы, кроме этих, отбрасываются. Все остальные элементы

можно поместить в новый массив *оператором расширения* (spread operator) (...), который мы рассмотрим в главе 6.

```
const arr = [1, 2, 3, 4, 5];
let [x, y, ...rest] = arr;
x;              // 1
y;              // 2
rest;           // [3, 4, 5]
```

В этом примере `x` и `y` получают первые два элемента массива, а переменная `rest` — все остальные (вы не обязаны называть переменную `rest`; вы можете использовать любое имя по своему усмотрению). Деструктуризация массива облегчает обмен значениями с переменными (ранее это требовало временной переменной).

```
let a = 5, b = 10;
[a, b] = [b, a];
a;              // 10
b;              // 5
```

Деструктуризация воздействует не только на массивы, но и на любой итерируемый объект (который мы рассмотрим в главе 9).

В этих простых примерах было бы проще присвоить переменные непосредственно, а не использовать деструктуризацию. Но деструктуризация пригодится, когда вы получаете объект или массив из внешних источников, и из него нужно быстро выбрать определенные элементы. Мы рассмотрим этот интересный эффект в главе 6.

Операторы объектов и массивов

Объекты, массивы и функции имеют коллекцию специальных операторов. Некоторые из них уже встречались выше (такие, как операторы доступа к члену и вычисляемого доступа к члену), а остальные будут описаны в главах 6, 8 и 9. Для полноты они приведены в табл. 5.10.

Таблица 5.10. Операторы объектов и массивов

Оператор	Описание	Глава
`.`	Доступ к члену	3
`[]`	Вычисляемый доступ к члену	3
`in`	Оператор проверки существования свойства	9
`new`	Оператор создания экземпляра объекта	9
`instanceof`	Оператор проверки цепи прототипов	9
`...`	Оператор расширения	6 и 8
`delete`	Оператор удаления	3

Выражения в строковых шаблонах

Строковые шаблоны (template string), с которыми мы познакомились в главе 3, применяются для введения значения *любого выражения* в строку. В примере из главы 3 использовался строковый шаблон для отображения текущей температуры. Что если мы захотим отобразить разницу температур или температуру в градусах Фаренгейта, а не Цельсия? Мы можем использовать выражения в строковых шаблонах.

```
const roomTempC = 21.5;
let currentTempC = 19.5;
const message = 'Температура на улице отличается от комнатной на ' +
    '${currentTempC-roomTempC}\u00b0C градуса.';
const fahrenheit =
    'Наружная температура ${currentTempC * 9/5 + 32}\u00b0F';
```

И снова мы видим приятную симметрию, которую привносят выражения. Мы можем использовать переменные в строковом шаблоне, поскольку переменная — это просто один из типов выражения.

Выражения и шаблоны управления потоком

В главе 4 мы рассмотрели несколько общих шаблонов управления потоком. Теперь, изучив ряд выражений, способных влиять на поток выполнения (тройственные выражения и вычисление по сокращенной схеме), мы можем рассмотреть несколько дополнительных шаблонов управления потоком.

Преобразование операторов `if...else` в условные выражения

Всякий раз, когда оператор `if...else` используется для возврата значения (либо в составе оператора присваивания, небольшого выражения или возвращаемого значения функции), предпочтительнее использовать условный оператор. В результате получается более компактный и читабельный код. Например, код

```
if(isPrime(n)) {
    label = 'prime';
} else {
    label = 'non-prime';
}
```

лучше написать так.

```
label = isPrime(n) ? 'prime' : 'non-prime';
```

Преобразование операторов `if` в сокращенные выражения логического ИЛИ

Подобно тому, как возвращающие значение операторы `if...else` можно легко преобразовать в условные выражения, возвращающие значение операторы `if` можно легко преобразовать в сокращенные выражения логического ИЛИ. Эта методика не столь очевидна, как условные операторы для операторов `if...else`, но вы будете их видеть очень часто, поэтому о них следует знать. Например, код

```
if(!options) options = {};
```

может быть легко преобразован в

```
options = options || {};
```

Заключение

В JavaScript, как и в большинстве современных языков программирования, есть обширная и полезная коллекция операторов, являющихся фундаментальными блоками для создания данных и манипулирования ими. Одни из них, такие как побитовые операторы, вы, вероятно, будете использовать очень редко. Другие, такие как операторы доступа к члену, вы будете использовать, даже не думая о них как об операторах (это может пригодиться только тогда, когда вы пытаетесь решить сложную задачу на приоритет операторов).

Операторы присваивания, арифметические, сравнения и логические весьма распространены, и вы будете часто их использовать, поэтому убедитесь в том, что вы хорошо в них разобрались перед продолжением чтения книги.

Функции

Функция (function) — это самодостаточный набор операторов, выполняющийся как единый блок; по существу, можно считать ее подпрограммой. Функции — это основа мощи и выразительности языка JavaScript. В данной главе описаны основы их применения и соответствующие механизмы.

У каждой функции есть *тело* (body) — набор составляющих функцию операторов.

```
function sayHello() {
    // это тело; оно начинается с открывающей фигурной скобки...

    console.log("Hello world!");
    console.log("!Hola mundo!");
    console.log("Hallo wereld!");
    console.log("Привет мир!");


    // ... и завершается закрывающей фигурной скобкой
}
```

Это пример *объявления функции* по имени `sayHello`. Само по себе объявление функции *не выполняет* ее тело: если вы попытаетесь запустить этот пример, то не увидите на консоли сообщения "`Hello world!`" на нескольких языках. Для *вызова* функции (называемого также *выполнением* или *запуском*) вы должны использовать имя функции, сопровождаемое круглыми скобками:

```
sayHello(); // "Hello, World!" выводится на консоль на разных языках
```

Термины *вызов (call)*, *выполнение (execute)* и *запуск (run)* являются синонимами, и я буду использовать их в книге все, чтобы они стали для вас привычными. В определенных контекстах и языках между этими терминами могут быть различия, но в повседневном употреблении они эквивалентны.

Возвращаемые значения

Вызов функции — это выражение, а как известно, выражения возвращают значения. Так что возвращает вызов функции? *Возвращаемое значение* (return value). Ключевое слово `return` в теле функции *немедленно завершает ее выполнение и возвращает определенное значение*, которое и является результатом вызова функции. Давайте изменим наш пример: вместо вывода на консоль просто возвратим приветствие.

```
function getGreeting() {
    return "Hello world!";
}
```

Теперь, когда происходит вызов, функция создает возвращаемое значение.

```
getGreeting();        // "Hello, World!"
```

Без явного оператора `return` возвращаемое значение будет `undefined`. Функция может возвратить значение любого типа; в качестве упражнения попробуйте создать функцию `getGreetings`, возвращающую массив, содержащий строки "Hello, World" на разных языках.

Вызов или обращение

Функции в JavaScript являются объектами и могут быть переданы и присвоены подобно любым другим объектам. Важно понимать различие между *вызовом* (calling) функции и просто *обращением* (referencing) к ней. Когда за идентификатором функции следуют круглые скобки, JavaScript знает, что осуществляется вызов: выполняется тело функции и выражение возвращает значение. Без круглых скобок вы просто обращаетесь к функции, как к любому другому значению, а это не вызов. Опробуйте следующее на консоли JavaScript.

```
getGreeting();       // "Hello, World!"
getGreeting;         // функция getGreeting()
```

Возможность обращения к функции, как к любому другому значению (не вызывая ее), обеспечивает бо́льшую гибкость языка. Например, вы можете присвоить функцию переменной, что позволит вызвать функцию под другим именем.

```
const f = getGreeting;
f();               // "Hello, World!"
```

Или можете присвоить функцию свойству объекта.

```
const o = {};
o.f = getGreeting;
o.f();             // "Hello, World!"
```

Или даже добавить функцию в массив.

```
const arr = [1, 2, 3];
arr[1] = getGreeting; // arr теперь [1, function getGreeting(), 2]
arr[1]();             // "Hello, World!"
```

Этот последний пример проясняет роль круглых скобок: если JavaScript встречает круглые скобки, которые следуют за значением, значение считается функцией и эту функцию следует вызвать. В приведенном примере `arr[1]` является выражением, которое возвращает значение. Данное значение, сопровождаемое круглыми скобками, является сигналом JavaScript о том, что это значение — функция и ее следует вызвать.

Если вы попытаетесь добавить круглые скобки к значению, не являющемуся функцией, то вы получите ошибку. Например, `"whoops"()` приведет к сообщению об ошибке `TypeError: "whoops" is not a function`.

Аргументы функции

Мы теперь знаем, как вызывать функции и получать значения *из* них. А как передавать информацию *в* них? Основной механизм передачи информации при вызове функции — это *аргументы* (argument) функции (иногда называемые ее *параметрами* (parameter)). Аргументы похожи на переменные, которые не существуют, пока функция не будет вызвана. Давайте рассмотрим функцию, получающую два числовых аргумента и возвращающую их средние значения.

```
function avg(a, b) {
    return (a + b)/2;
}
```

В этом объявлении функции `a` и `b` — это *формальные аргументы* (formal argument). При вызове функции формальные аргументы получают значения и становятся *фактическими аргументами* (actual argument).

```
avg(5, 10);     // 7.5
```

В этом примере формальные аргументы `a` и `b` получают значения `5` и `10` и становятся фактическими аргументами (которые очень похожи на переменные, но специфичны для тела функции).

У новичков нередко возникает недопонимание того, что аргументы *существуют только в функции*, даже если у них то же имя, что и у переменных за пределами функции. Рассмотрим пример.

```
const a = 5, b = 10;
avg(a, b);
```

Переменные `a` и `b` здесь являются отдельными переменными, независимыми от *аргументов* `a` и `b` функции `avg`, несмотря на совпадение их имен. Когда вы

вызываете функцию, ее аргументы получают *значения*, которые вы передаете, а не сами переменные. Рассмотрим следующий код.

```
function f(x) {
    console.log('Внутри f: x=${x}');
    x = 5;
    console.log('Внутри f: x=${x} (после присваивания)');
}

let x = 3;
console.log('Перед вызовом f: x=${x}');
f(x);
console.log('После вызова f: x=${x}');
```

Если вы запустите этот пример, то увидите

```
Перед вызовом f: x=3
Внутри f: x=3
Внутри f: x=5 (после присваивания)
После вызова f: x=3
```

Здесь важно то, что присваивание значения `x` в функции не затрагивает переменную `x` вне функции; дело в том, что это две разные сущности, имена которых случайно совпали.

Всякий раз, когда мы присваиваем аргумент функции, это никак не влияет ни на какие переменные за пределами функции. Тем не менее вполне возможно изменить *тип объекта* (object type) в функции способом, который изменит сам объект, видимый за пределами функции.

```
function f(o) {
    o.message = 'Изменено в f (предыдущее значение: '${o.message}')';
}
let o = {
    message: "Начальное значение"
};
console.log('Перед вызовом f: o.message="${o.message}"');
f(o);
console.log('После вызова f: o.message="${o.message}"');
```

Это даст следующий результат.

```
Перед вызовом f: o.message="Начальное значение"
После вызова f: o.message="Изменено в f (предыдущее значение:
'Начальное значение')"
```

В этом примере мы видим, что внутри функции `f` был изменен объект `o`, и эти изменения повлияли на объект `o` за пределами функции. Это выявляет основное различие между базовыми типами и объектами. Базовые типы не могут быть изменены (можно изменить только значение переменной базового типа, но само базовое *значение* при этом не изменяется). Объекты, напротив, могут быть изменены.

Для ясности: объект o в функции не зависит от объекта o за пределами функции, но оба они *обращаются к одному и тому же объекту*. Мы можем увидеть это различие снова в присваивании.

```
function f(o) {
    o.message = "Изменено в f";
    o = {
        message: "Новый объект!"
    };
console.log('Внутри f: o.message="${o.message}" (после присваивания)');
}
let o = {
    message: 'Начальное значение'
};
console.log('Перед вызовом f: o.message="${o.message}"');
f(o);
console.log('После вызова f: o.message="${o.message}"');
```

Если запустить этот пример, то можно увидеть

```
Перед вызовом f: o.message="Начальное значение"
Внутри f: o.message="Новый объект!" (после присваивания)
После вызова f: o.message=" Изменено в f"
```

Ключом к пониманию происходящего здесь является то, что *аргумент* o (в функции) — это вовсе не переменная o за пределами функции. При вызове `f` оба указывают на один и тот же объект, но когда o присваивается объект внутри `f`, оно начинает указывать на *новый, независимый* объект, а o вне функции все еще продолжает указывать на первоначальный объект.

Базовые типы в JavaScript считаются *типами значения* (value type), как и во всех других языках программирования, поскольку при передаче по значению они *копируются*. Объекты называют *ссылочными типами* (reference type), поскольку при их передаче по ссылке обе переменные продолжают ссылаться на *один и тот же объект* (т.е. обе они содержат ссылку на тот же объект).

Определяют ли аргументы функцию?

Во многих языках в *сигнатуру* (signature) функции включаются ее аргументы. Например, в языке C функция `f()` (без аргументов) отличается от функции `f(x)` (с одним аргументом), которая в свою очередь отличается от функции `f(x, y)` (с двумя аргументами). В языке JavaScript такого различия нет, и когда у вас есть функция по имени `f`, можете вызывать ее без аргументов, с одним аргументом или с десятью аргументами (причем вы всегда вызовете одну и ту же функцию).

Таким образом, вы можете вызвать любую функцию с любым количеством аргументов. Если при вызове функции вы не укажете аргументы, то им неявно присваивается значение `undefined`.

```
function f(x) {
    return 'Внутри f: x=${x}';
}
f(); // "Внутри f: x=undefined"
```

Далее в этой главе мы увидим, как справиться с ситуацией, в которой передается больше аргументов, чем определено в функции.

Деструктуризация аргументов

Подобно деструктурирующему присваиванию переменных (см. главу 5), существуют также *деструктурированные аргументы* (destructured argument) функций. В конце концов, аргументы очень похожи на переменные! Рассмотрим деструктуризацию объекта на индивидуальные переменные.

```
function getSentence({ subject, verb, object }) {
    return '${subject} ${verb} ${object}';
}

const o = {
    subject: "I",
    verb: "love",
    object: "JavaScript",
};

getSentence(o);    // "I love JavaScript"
```

Как и при деструктурирующем присваивании, имена свойств должны быть строковыми идентификаторами, а переменным, которым не соответствуют свойства в исходном объекте, назначается значение `undefined`.

Можно также деструктурировать массив.

```
function getSentence([ subject, verb, object ]) {
    return '${subject} ${verb} ${object}';
}

const arr = [ "I", "love", "JavaScript" ];
getSentence(arr);    // "I love JavaScript"
```

И наконец вы можете использовать оператор расширения (`...`) для сбора любых дополнительных аргументов.

```
function addPrefix(prefix, ...words) {
    // позже мы изучим лучший способ сделать это!
```

```
    const prefixedWords = [];
    for(let i=0; i<words.length; i++) {
        prefixedWords[i] = prefix + words[i];
    }
    return prefixedWords;
}

addPrefix("con", "verse", "vex"); // ["converse", "convex"]
```

Обратите внимание: если вы используете оператор расширения в объявлении функции, то это *должен быть последний аргумент*. Если вы поместите аргументы после него, у JavaScript не будет способа выяснить, что должно войти в расширение аргумента, а что должно войти в остальные аргументы.

В ES5 подобные функциональные возможности реализуются с помощью специальной переменной `arguments`, которая существует только в пределах тела функции. Эта переменная была не фактическим массивом, а "подобным массиву" объектом, который зачастую требовал специальной обработки или преобразования в надлежащий массив. Расширение аргументов в ES6 ликвидирует этот недостаток и ему должно отдаваться предпочтение перед использованием переменной `arguments` (которая все еще доступна).

Стандартные аргументы

Нововведением ES6 является способность определять *стандартные значения* (default value) для аргументов. Обычно, когда значения для аргументов не заданы, им присваивается значение `undefined`. С помощью стандартных значений можно определить некое другое значение для тех аргументов, которые не были переданы в функцию.

```
function f(a, b = "default", c = 3) {
    return '${a} - ${b} - ${c}';
}

f(5, 6, 7);   // "5 - 6 - 7"
f(5, 6);      // "5 - 6 - 3"
f(5);         // "5 - default - 3"
f();          // "undefined - default - 3"
```

Функции как свойства объектов

Когда функция является свойством объекта, ее зачастую называют *методом* (method), чтобы отличать от обычной функции (вскоре мы узнаем больше различий между *функцией* и *методом*). Мы уже видели в главе 3, как можно добавить

функцию в существующий объект. Мы можем также добавить метод в объект непосредственно.

```
const o = {
    name: 'Wallace',                        // свойство базового типа
    bark: function() { return 'Woof!'; }, // свойство функция (метод)
}
```

В спецификацию ES6 введен новый сокращенный синтаксис для методов. Следующее функционально эквивалентно предыдущему примеру.

```
const o = {
    name: 'Wallace',            // свойство базового типа
    bark() { return 'Woof!'; }, // свойство функции (метод)
}
```

Ключевое слово `this`

В теле функции доступна предназначенная только для чтения специальная переменная `this`. Это ключевое слово обычно ассоциируется с объектно-ориентированным программированием, и мы узнаем больше о его использовании в главе 9. В JavaScript, однако, оно используется несколькими способами.

Обычно ключевое слово `this` имеет отношение к функциям, являющимся свойствами объектов. При вызове метода переменной `this` присваивается значение конкретного объекта, в котором произошел вызов.

```
const o = {
    name: 'Wallace',
    speak() { return 'Меня зовут ${this.name}!'; },
}
```

Когда происходит вызов `o.speak()`, переменная `this` будет связана с объектом `o`.

```
o.speak();    // "Меня зовут Wallace!"
```

Важно понимать, что `this` связывается согласно тому, *как функция вызвана, а не где объявлена*. Таким образом, `this` связана с `o` не потому, что `speak` — это свойство `o`, а потому, что мы вызвали этот метод непосредственно из `o` (`o.speak`). Давайте рассмотрим, что будет, если мы присвоим ту же функцию переменной.

```
const speak = o.speak;
speak === o.speak;    // true; обе переменные обращаются к той же функции
speak();              // "Меня зовут !"
```

В связи с другим способом вызова функции движок JavaScript не знает, что функция была первоначально объявлена в `o`; таким образом, переменной `this` было присвоено значение `undefined`.

Когда вы вызываете функцию таким способом, при котором привязка переменной `this` не очевидна (как при вызове метода `speak` ранее), становится довольно сложно предсказать ее значение. Все зависит от того, находитесь ли вы в строгом режиме, и от места, из которого функция вызывается. Мы преднамеренно не рассматриваем эти подробности, поскольку таких ситуаций лучше избегать. Чтобы узнать больше, обратитесь к теме *Code formatting* (форматирование кода) в документации MDN.

Термин *метод* (method) традиционно ассоциируется с объектно-ориентированным программированием, и в этой книге мы будем использовать его для обозначения функции, которая является свойством объекта и предназначена для вызова непосредственно из экземпляра объекта (например, `o.speak()`). Если в функции не используется переменная `this`, то мы все равно будем называть ее функцией независимо от того, где она объявлена.

Одна из особенностей переменной `this` зачастую сбивает с толку, когда необходимо получить доступ к ней во вложенной функции. Рассмотрим следующий пример, в котором мы используем в методе вспомогательную функцию.

```
const o = {
    name: 'Julie',
    greetBackwards: function() {
        function getReverseName() {
            let nameBackwards = '';
            for(let i=this.name.length-1; i>=0; i--) {
                nameBackwards += this.name[i];
            }
            return nameBackwards;
        }
        return '${getReverseName()} si eman ym ,olleH';
    },
};
o.greetBackwards();
```

Здесь мы используем вложенную функцию, `getReverseName`, для изменения имени на обратное. К сожалению, `getReverseName` не будет работать так, как ожидалось: при вызове `o.greetBackwards()` JavaScript привязывает переменную `this`, как и ожидалось к объекту `o`. Но когда происходит вызов `getReverseName` внутри функции `greetBackwards`, переменная `this` связывается с чем-то другим[1]. Обычно решение этой проблемы подразумевает применение второй переменной, которой присваивается значение `this`.

[1] Она будет связана с глобальным объектом или `undefined`, в зависимости от того, находитесь ли вы в строгом режиме. Мы не рассматриваем здесь все подробности, поскольку такой ситуации следует избегать.

```
const o = {
    name: 'Julie',
    greetBackwards: function() {
        const self = this;
        function getReverseName() {
            let nameBackwards = '';
            for(let i=self.name.length-1; i>=0; i--) {
                nameBackwards += self.name[i];
            }
            return nameBackwards;
        }
        return '${getReverseName()} si eman ym ,olleH';
    },
};
o.greetBackwards();
```

Это общепринятая методика, и вы часто будете встречать присваивание значения переменной `this` константам `self`, или `that`. Стрелочные функции, которые мы будем рассматривать далее в этой главе, являются еще одним средством решения этой проблемы.

Функциональные выражения и анонимные функции

До сих пор мы имели дело исключительно с *объявлениями функций* (function declaration), которые присваивают функции и тело (т.е. то, что функция делает), и идентификатор (он позволяет впоследствии вызывать функцию по имени). JavaScript поддерживает также *анонимные функции* (anonymous function), у которых не обязательно есть идентификатор.

У вас может возникнуть резонный вопрос "Как использовать функцию, у которой нет идентификатора? Как мы должны вызывать ее без идентификатора?" Ответ кроется в понятии *функциональных выражений* (function expression). Известно, что выражение — это нечто, что вычисляет значение, и мы также знаем, что функция — это также значение, как и все остальное в JavaScript. Функциональное выражение — это просто средство для того, чтобы объявить (возможно, безымянную) функцию. Функциональное выражение может быть присвоено чему-нибудь (в результате ему будет назначен идентификатор) или сразу же вызвано[2].

Функциональные выражения синтаксически идентичны объявлениям функций, за исключением того, что вы можете опустить имя функции. Давайте рассмотрим пример, в котором мы используем функциональное выражение и присвоим результат переменной (который фактически эквивалентен объявлению функции).

[2] Это *немедленно вызываемое функциональное выражение* (Immediately Invoked Function Expression — IIFE), которое мы рассмотрим в главе 7.

```
const f = function() {
    // ...
};
```

Результат такой, как будто мы объявили функцию обычным способом: здесь имеется идентификатор `f`, который позволяет обращаться к функции. Как и при обычном объявлении функции, мы можем вызвать ее, используя синтаксис `f()`. Единственное отличие в том, что здесь мы создаем анонимную функцию (используя функциональное выражение) и присваиваем ее переменной.

Анонимные функции используются все время: как аргументы других функций или методов либо при создании функциональных свойств в объекте. Мы будем встречаться с ними повсюду в книге.

Я сказал, что имя функции является *необязательным* в функциональном выражении... Так что же происходит, когда мы присваиваем функции имя *и* присваиваем ее переменной (и для чего это может нам понадобиться)? Рассмотрим пример.

```
const g = function f() {
    // ...
}
```

Когда функция создается таким способом, имя `g` имеет приоритет, и для обращения к функции (извне функции) мы используем имя `g`; попытка доступа к `f` приводит к ошибке неопределенности переменной. С учетом вышесказанного зачем бы нам это могло понадобиться? Это может быть необходимо, если нужно обратиться к функции из самой функции (так называемая *рекурсия* (recursion)).

```
const g = function f(stop) {
    if(stop) console.log('f остановлена');
    f(true);
};
g(false);
```

В самой функции мы используем `f`, чтобы сослаться на функцию, а вне мы используем `g`. Нет никакого особенно серьезного основания, чтобы присваивать функции два разных имени, но здесь мы делаем так для пояснения работы именованных функциональных выражений.

Поскольку объявление функции и функционального выражения выглядят идентично, вы могли бы задаться вопросом «Как JavaScript их различает (или есть ли там какое-нибудь различие)?» Ответ — *контекст*: если объявление функции используется как выражение, то это функциональное выражение, а если нет, то это объявление функции.

Различие является главным образом академическим, и вы не должны обычно думать о нем. Определяя именованную функцию, которую вы намереваетесь вызвать позже, вы, вероятно, будете использовать объявление функции, не думая об этом, а если необходимо создать функцию для присваивания чему-то или для передачи в другую функцию, вы будете использовать функциональное выражение.

Стрелочная нотация

В спецификацию ES6 введен новый долгожданный синтаксис *стрелочной нотации* (arrow notation). Это чрезвычайно полезный синтаксис (он имеет одно серьезное функциональное отличие, до которого мы вскоре дойдем), который существенно экономит время на вводе слова `function`, а также сокращает количество фигурных скобок, которые нужно ввести.

Стрелочные функции позволяют упростить синтаксис тремя способами.

- Опустить слово `function`.
- Если функции передается один аргумент, опустить круглые скобки.
- Если тело функции — одно выражение, опустить фигурные скобки и оператор `return`.

Стрелочные функции всегда являются анонимными. Вы вполне можете присвоить их переменной, но не можете создать именованную функцию, как в случае использования ключевого слова `function`.

Рассмотрим следующие эквивалентные функциональные выражения.

```
const f1 = function() { return "hello!"; }
// ИЛИ
const f1 = () => "hello!";

const f2 = function(name) { return 'Hello, ${name}!'; }
// ИЛИ
const f2 = name => 'Hello, ${name}!';

const f3 = function(a, b) { return a + b; }
// ИЛИ
const f3 = (a,b) => a + b;
```

Эти примеры немного надуманы; обычно, если необходима именованная функция, вы просто используете обычное объявление функции. Стрелочные функции особенно полезны для создания и передачи в виде параметров анонимных функций, которые мы будем видеть весьма часто начиная с главы 8.

У стрелочных функций действительно есть одно серьезное отличие от обычных функций: переменная `this` привязывается лексически, точно так же, как и любая другая переменная. Вспомните наш пример `greetBackwards` ранее в главе. Со стрелочной функцией мы можем использовать переменную `this` во внутренней функции.

```
const o = {
    name: 'Julie',
    greetBackwards: function() {
        const getReverseName = () => {
            let nameBackwards = '';
```

```
            for(let i=this.name.length-1; i>=0; i--) {
                nameBackwards += this.name[i];
            }
            return nameBackwards;
        };
        return '${getReverseName()} si eman ym ,olleH';
    },
};
o.greetBackwards();
```

У стрелочных функций есть еще два дополнительных отличия от обычных функций: они не могут использоваться как конструкторы объекта (см. главу 9) и в них недоступна специальная переменная `arguments` (в которой больше нет необходимости благодаря оператору расширения).

Методы `call`, `apply` и `bind`

Мы уже видели, что значение переменной `this` зависит от вызываемого контекста (как и в других объектно-ориентированных языках). Но JavaScript позволяет определять, к чему привязана переменная `this`, независимо от того, как или где вызывается рассматриваемая функция. Давайте начнем с метода `call`, который доступен во всех функциях. Он позволяет вызывать функцию с определенным значением `this`.

```
const bruce = { name: "Bruce" };
const madeline = { name: "Madeline" };

// эта функция не связана ни с каким объектом,
// но все же в ней используется 'this'!
function greet() {
    return 'Привет! Меня зовут ${this.name}!';
}

greet();               // "Привет! Меня зовут !" - 'this' не привязана
greet.call(bruce);     // "Привет! Меня
зовут Bruce!" - 'this' привязана к 'bruce'
greet.call(madeline); // "Привет! Меня зовут Madeline!" - 'this' привязана
                       // к 'madeline'
```

Как можно заметить, метод `call` позволяет вызывать функцию, как будто это метод, предоставляемый объектом, к которому привязана переменная `this`. Первый аргумент метода `call` — это значение, к которому вы хотите привязать `this`, а все остальные аргументы становятся аргументами вызываемой функции.

```
function update(birthYear, occupation) {
    this.birthYear = birthYear;
    this.occupation = occupation;
}
```

```
update.call(bruce, 1949, 'singer');
    // bruce теперь { name: "Bruce", birthYear: 1949,
    //    occupation: "singer" }
update.call(madeline, 1942, 'actress');
    // madeline теперь { name: "Madeline", birthYear: 1942,
    //    occupation: "actress" }
```

Метод `apply` идентичен методу `call`, за исключением способа, которым он обрабатывает аргументы функции. Методу `call` аргументы передаются непосредственно, точно так же, как и обычной функции. Метод `apply` аргументы передаются в виде массива.

```
update.apply(bruce, [1955, "actor"]);
    // bruce теперь { name: "Bruce", birthYear: 1955,
    //    occupation: "actor" }
update.apply(madeline, [1918, "writer"]);
    // madeline теперь { name: "Madeline", birthYear: 1918,
    //    occupation: "writer" }
```

Метод `apply` полезен, если у вас есть массив и вы хотите использовать его значения как аргументы функции. Классический пример — поиск минимального или максимального числа в массиве. Вспомогательным функциям `Math.min` и `Math.max` можно передать любое количество аргументов, а они возвращают минимальное или максимальное значение, соответственно. Мы можем использовать метод `apply`, чтобы вызвать эти функции с существующим массивом.

```
const arr = [2, 3, -5, 15, 7];
Math.min.apply(null, arr);       // -5
Math.max.apply(null, arr);       // 15
```

Обратите внимание, что вместо значения `this` мы просто передаем `null`. Так происходит потому, что в функциях `Math.min` и `Math.max` не используется переменная `this` вообще; поэтому не имеет никакого значения, что мы передаем в качестве значения для `this`.

Оператор расширения (`...`) ES6 позволяет достичь того же результата, что и метод `apply`. В экземпляре нашего метода `update`, в котором значение `this` действительно важно, мы все еще должны использовать метод `call`, но для функций `Math.min` и `Math.max`, для которых оно не имеет значения, мы можем использовать оператор расширения, чтобы вызвать эти функции непосредственно.

```
const newBruce = [1940, "martial artist"];
update.call(bruce, ...newBruce); // эквивалент apply(bruce, newBruce)
Math.min(...arr);                // -5
Math.max(...arr);                // 15
```

Есть еще одна функция, `bind`, которая позволяет определить значение для переменной `this`. Функция `bind` позволяет *перманентно* ассоциировать значение для `this` с функцией. Предположим, что мы распространяем свой метод `update`

и хотим удостовериться, что в нем переменной `this` *всегда* будет присвоено значение `bruce`, независимо от того, как он будет вызван (даже с функцией `call`, `apply` или другой функцией `bind`). Функция `bind` позволяет сделать так.

```
const updateBruce = update.bind(bruce);

updateBruce(1904, "actor");
  // bruce теперь { name: "Bruce", birthYear: 1904, occupation: "actor" }
updateBruce.call(madeline, 1274, "king");
  // bruce теперь { name: "Bruce", birthYear: 1274, occupation: "king" };
  // madeline не было присвоено!
```

Тот факт, что действие функции `bind` является постоянным, делает ее потенциальным источником ошибок, которые трудно обнаружить: в результате вы получаете функцию, которую фактически нельзя использовать с функциями `call`, `apply` или `bind` (во второй раз). Представьте себе, что функция передается в виде параметра, и она, будучи вызванной с помощью функций `call` или `apply` в некоем отдаленном месте, полностью уверена в правильной привязке `this`. Я не говорю вам, что `bind` не нужно использовать, она весьма полезна, но помните о налагаемых ею ограничениях.

Вы можете также предоставить функции `bind` параметры, которые позволяют создать новую функцию, всегда вызываемую с определенными параметрами. Например, если необходима функция `update`, которая всегда устанавливает год рождения `bruce` равным `1949`, но все еще позволяет изменять род занятия, вполне можете поступить следующим образом.

```
const updateBruce1949 = update.bind(bruce, 1949);
updateBruce1949("singer, songwriter");
    // bruce теперь { name: "Bruce", birthYear: 1949,
    //    occupation: "singer, songwriter" }
```

Заключение

Функции — это жизненно важная часть JavaScript. Они делают намного больше, чем просто собирают код в блоки: они позволяют создавать невероятно мощные алгоритмические модули. Эта глава была посвящена в основном механике функций — сухое, но важное введение. Обладая этой информацией, нам будет проще изучать функции в следующих главах.

Область видимости

Область видимости (scope) устанавливает, где и когда определяются переменные, константы и аргументы. Мы уже имеем некоторое представление об области видимости: нам известно, что аргументы функции существуют только в теле функции. Рассмотрим следующее.

```
function f(x) {
    return x + 3;
}
f(5);     // 8
x;        // ReferenceError: x не определена
```

Мы знаем, что переменная `x` существует очень недолго (только при вычислении `x + 3`), а за пределами тела функции `x` как будто не существует. Таким образом, мы говорим, что *область видимости* переменной `x` — это функция `f`.

Когда мы говорим, что область видимости переменной — это данная функция, мы должны помнить, что формальных аргументов в теле функции не существует, пока функция не будет вызвана (и они не станут, таким образом, фактическими аргументами). Функция может быть вызвана многократно, и при каждом вызове ее аргументы появляются и затем выходят из области видимости, когда функция завершается.

Мы также принимаем как очевидное, что переменные и константы не существуют, пока мы их не создадим. Таким образом, они не находятся в области видимости, пока мы не объявляем их с ключевым словом `let` или `const` (`var` — это частный случай, который мы рассмотрим далее в этой главе).

В некоторых языках есть явное различие между *объявлением* (declaration) и *определением* (definition). Как правило, объявление переменной означает, что вы объявляете о ее существовании, указав компилятору ее идентификатор. Определение, напротив, обычно означает объявление *и* присвоение значения переменной. В JavaScript эти два термина являются синонимами, поскольку всем переменным присваиваются значения при их объявлении (в противном случае им неявно присваиваются значения `undefined`).

Область видимости и существование переменных

Интуитивно понятно, что если переменной не существует, ее нет и в области видимости. Таким образом, переменные, которые еще не были объявлены, или переменные, которые прекратили существование после выхода из функции, очевидно не находятся в области видимости.

А наоборот? Если переменная не находится в области видимости, то означает ли это, что ее не существует? Вовсе не обязательно, и именно здесь необходимо сделать различие между *областью видимости* (scope) и *существованием* (existence) переменной.

Область видимости (или *видимость* (visibility)) относится к идентификаторам, которые в настоящее время видимы и доступны выполняющейся в данный момент части программы (называемой *контекстом выполнения* (execution context)). Существование, напротив, относится к идентификаторам, которые содержат нечто, для чего была *распределена* (т.е. *зарезервирована*) область памяти. Скоро мы увидим примеры переменных, которые существуют, но не находятся в области видимости.

Когда нечто прекращает существовать, JavaScript не обязательно освобождает память сразу же: она просто помечается как не используемая и освобождается только при периодически запускаемом процессе *сборки мусора* (garbage collection). Сборка мусора в JavaScript осуществляется автоматически и будет вас интересовать только в определенных чрезвычайно требовательных приложениях.

Лексическая или динамическая область видимости

Глядя на исходный код программы, вы видите ее *лексическую структуру* (lexical structure). Когда программа выполняется фактически, поток выполнения может быть не последовательным. Рассмотрим программу с двумя функциями.

```
function f1() {
    console.log('one');
}
function f2() {
    console.log('two');
}

f2();
f1();
f2();
```

Лексически эта программа — просто набор операторов, которые мы обычно читаем сверху вниз. Но когда мы запускаем эту программу, поток выполнения переходит сначала к телу функции `f2`, затем — к телу функции `f1` (даже при том, что она определена до `f2`) и наконец — снова к телу функции `f2`.

Области видимости в JavaScript являются *лексическими*, а значит, можно определить, какие переменные находятся в области видимости, просто посмотрев на исходный код. Я не хочу сказать, что области видимости всегда и сразу *очевидны* из исходного кода: в этой главе мы увидим несколько примеров, которые требуют пристального внимания для определения областей видимости.

Лексическая область видимости означает, что в области видимости функции находятся только те переменные, которые были определены до *момента определения* самой функции (не путать с моментом ее вызова). Рассмотрим пример.

```
const x = 3;
function f() {
    console.log(x); // это сработает
    console.log(y); // а это — нет
}

const y = 3;
f();
```

Когда мы определили функцию `f` переменная `x` уже существовала, а переменная `y` — еще нет. Затем мы объявили `y` и вызвали `f`. Переменная `x` находится в области видимости тела функции `f` при ее вызове, а переменная `y` — нет. Это пример лексической области видимости: у функции `f` есть доступ к идентификаторам, которые существовали на момент ее *определения*, но не на момент *вызова*.

Лексическими в JavaScript являются *глобальная область видимости* (global scope), *область видимости блока* (block scope) и *области видимости функции* (function scope).

Глобальная область видимости

Область видимости имеет иерархический, древовидный характер. Та область видимости, в которой вы находитесь в момент запуска программы, называется *глобальной областью видимости* (global scope). Вновь запущенная программа JavaScript (прежде, чем будут вызваны любые функции) выполняется в глобальной области видимости. Таким образом, все, что вы объявите в глобальной области видимости, будет доступно для всех областей видимости в вашей программе.

Все, что объявлено в глобальной области видимости, называется *глобальными переменными*, а у глобальных переменных, как известно, очень плохая репутация. Открыв любую книгу по программированию вы узнаете, что при использовании глобальных переменных "земля уйдет из под ваших ног и поглотит вас целиком". Так почему же глобальные переменные столь плохи?

Глобальные переменные вовсе не плохи, это нужная вещь. Плохо, когда глобальную область видимости используют *неправильно*. Мы уже упоминали, что все доступное в глобальной области видимости доступно во *всех* областях видимости. Отсюда мораль: глобальные переменные следует использовать рассудительно.

Догадливый читатель мог бы подумать: "Хорошо, я создам в глобальной области видимости одну функцию и сведу мои глобальные переменные к одной функции!" Прекрасно, но только теперь вы просто перенесли проблему на один уровень вниз. Все, что будет объявлено в пределах этой функции, будет доступно для всего, что вызывается в этой функции... что едва ли лучше глобальной области видимости!

Подведем черту: у вас, вероятно, будет нечто в глобальной области видимости, и это не обязательно плохо, а вот пытаться избегать следует того, что *зависит* от глобальной области видимости. Давайте рассмотрим простой пример: отслеживание информации о пользователе. Ваша программа отслеживает имя и возраст пользователя, а также имеет несколько функций, которые работают с этой информацией. Это можно сделать, используя глобальные переменные.

```
let name = "Irena";      // глобальная
let age = 25;            // глобальная

function greet() {
    console.log('Hello, ${name}!');
}
function getBirthYear() {
    return new Date().getFullYear() - age;
}
```

Проблема этого подхода в том, что наши функции жестко зависят от контекста (или области видимости), из которого они вызываются. Любая функция (в любой части вашей программы) может изменить значение `name` (случайно или преднамеренно). Идентификаторы "`name`" и "`age`" (имя и возраст) весьма распространены и вполне могут использоваться в другом месте по другим причинам. Поскольку функции `greet` и `getBirthYear` зависят от глобальных переменных, они, возможно, безосновательно полагают, что остальная часть программы использует `name` и `age` правильно.

Лучше поместить всю информацию о пользователе в один объект.

```
let user = {
    name = "Irena",
    age = 25,
};

function greet() {
    console.log('Hello, ${user.name}!');
}
function getBirthYear() {
    return new Date().getFullYear() - user.age;
}
```

В этом простом примере мы сократили количество идентификаторов в глобальной области видимости только на один (мы избавились от `name` и `age`, но добавили `user`), но что если у нас будет 10 пользователей... или 100?

Но мы могли бы добиться большего: наши функции `greet` и `getBirthYear` все еще зависят от глобального объекта `user`, который может быть изменен как-то еще. Давайте улучшим эти функции таким образом, чтобы они не зависели от глобальной области видимости.

```
function greet(user) {
    console.log('Hello, ${user.name}!');
}
function getBirthYear(user) {
    return new Date().getFullYear() - user.age;
}
```

Теперь наши функции могут быть вызваны из *любой* области видимости, и пользователь им передается явно (узнав о модулях и объектно-ориентированном программировании, мы увидим лучшие способы решения этой проблемы).

Если бы все программы были так просты, то едва ли имело бы значение, используем мы глобальные переменные или нет. Но когда ваша программа насчитывает тысячи (или сотни тысяч) строк, вы не можете помнить все области видимости сразу (или даже только те, что находятся на экране). Поэтому становится критически важно не зависеть от глобальной области видимости.

Область видимости блока

Ключевые слова `let` и `const` объявляют идентификаторы в *области видимости блока* (block scope). В главе 5 упоминалось, что блок — это список операторов, заключенный в фигурные скобки. Поэтому, под областью видимости блока подразумеваются только те идентификаторы, которые доступны в пределах блока.

```
console.log('перед блоком');
{
    console.log('внутри блока');
    const x = 3;
    console.log(x): // выводит 3
}
console.log('за пределами блока; x=${x}'); // ReferenceError: x не определена
```

Это *автономный блок* (standalone block): обычно блок является частью оператора управления потоком, такого как `if` или `for`, но это вполне допустимый синтаксис, блок может быть и автономным. Переменная `x` определяется в блоке, поэтому по завершении блока она выходит из области видимости и считается неопределенной.

В главе 4 упоминалось, что автономные блоки не очень часто применяются на практике; они используются для контроля областей видимости (как мы увидим в этой главе), но это редко необходимо. Однако они очень удобны для объяснения действия областей видимости, вот почему мы используем их в этой главе.

Маскировка переменной

Популярным источником недопонимания являются одноименные переменные или константы в разных областях видимости. Когда области видимости следуют одна за другой, все относительно просто.

```
{
    // блок 1
    const x = 'blue';
    console.log(x);     // выводит "blue"
}
console.log(typeof x); // выводит "undefined"; x вне области видимости
{
    // блок 2
    const x = 3;
    console.log(x);     // выводит "3"
}
console.log(typeof x); // выводит "undefined"; x вне области видимости
```

Здесь вполне понятно, что есть две разные переменные, обе по имени x, но в разных областях видимости. Теперь рассмотрим, что происходит во *вложенных* областях видимости.

```
{
    // внешний блок
    let x = 'blue';
    console.log(x);      // выводит "blue"
    {
        // внутренний блок
        let x = 3;
        console.log(x); // выводит "3"
    }
    console.log(x);      // выводит "blue"
}
console.log(typeof x);  // выводит "undefined"; x вне области видимости
```

Этот пример демонстрирует *маскировку переменной* (variable masking). Переменная x во внутреннем блоке отличается от таковой во внешнем блоке (хотя и имеет такое же имя), что в действительности *маскирует* (или скрывает), переменную x, определенную во внешней области видимости.

Здесь важно понимать, что, когда процесс выполнения входит во внутренний блок и определяется новая переменная x, *в области видимости находятся обе переменные*; у нас просто нет никакого способа обратиться к переменной из внешней области видимости (поскольку у нее то же имя). Сравните это с предыдущим примером, где один x входит в область видимости, а затем выходит из нее прежде, чем вторая переменная x делает то же самое.

С учетом этого рассмотрим следующий пример.

```
{
    // внешний блок
    let x = { color: "blue" };
    let y = x;                  // y и x ссылаются на тот же объект
    let z = 3;
    {
        // внутренний блок
        let x = 5;              // внешний x теперь замаскирован
        console.log(x);         // выводит 5
        console.log(y.color); // выводит "blue"; объект, на который
                                // указывает y (и x во внешней области
                                // видимости), все еще находится в области
                                // видимости
        y.color = "red";
        console.log(z);         // выводит 3; z не замаскирована
    }
    console.log(x.color);       // выводит "red"; объект изменяется во
                                // внутренней области видимости
    console.log(y.color);       // выводит "red"; x и y ссылаются на тот
                                // же объект
    console.log(z);             // выводит 3
}
```

Маскировку переменных иногда называют *затенением* (shadowing) (т.е. переменная из внешней области видимости попадает в *тень* переменной с тем же именем во внутренней области). Мне никогда не нравился этот термин, поскольку тени никогда не скрывают сущности полностью, только делают их более темными. Когда переменная маскируется, она полностью недоступна.

Иерархический характер областей видимости на настоящий момент должен быть уже понятен: вы можете ввести новую область видимости, не покидая прежней. Это создает *цепь областей видимости*, определяющих, какие переменные находятся в области видимости: все переменные в текущей цепи областей видимости находятся в области видимости и (пока они не маскируются) вполне доступны.

Функции, замкнутые выражения и лексическая область видимости

До сих пор мы имели дело только с блоками, которые улучшают наглядность лексической области видимости, особенно если вы выравниваете свои блоки отступами. Функции, напротив, могут быть определены в одном месте,

а использованы — в другом, поэтому вам, возможно, придется приложить некоторые усилия, чтобы разобраться в их областях видимости.

В "традиционной" программе все ваши функции могли бы быть определены в глобальной области видимости, и если вы не будете обращаться к глобальной области видимости из своих функций (как я рекомендую), вы даже не должны будете заботиться о том, к какой области видимости у ваших функций есть доступ.

В современных программах на JavaScript, однако, функции зачастую определяются везде, где они необходимы. Их присваивают переменным или свойствам объектов, добавляют в массивы, передают в другие функции, возвращают из функций, а иногда не присваивают им имен вообще.

Довольно часто функцию преднамеренно определяют в некоторой области видимости, чтобы гарантированно получить доступ к переменным из этой области. В результате получается *замкнутое выражение* (closure) (вы можете считать его областью видимости, замкнутой вокруг функции). Давайте рассмотрим пример замкнутого выражения.

```
let globalFunc;                  // неопределенная глобальная функция
{
    let blockVar = 'a';          // переменная области видимости блока
    globalFunc = function() {
        console.log(blockVar);
    }
}
globalFunc();                    // выводит "a"
```

В функции `globalFunc` присваивается значение переменной в пределах блока: этот блок (и его родительская, глобальная область видимости) формирует замкнутое выражение. Независимо от того, где вы вызываете функцию `globalFunc`, у нее будет доступ к идентификаторам в этом замкнутом выражении.

Давайте рассмотрим происходящее: при вызове у функции `globalFunc` есть доступ к переменной `blockVar`, *несмотря на то что мы вышли из этой области видимости*. Обычно после выхода из области видимости объявленные в этой области видимости переменные могут безопасно прекратить существование. Здесь движок JavaScript замечает, что функция определена в этой области видимости (обратиться к функции можно и за пределами области видимости), поэтому переменная должна быть всегда доступна.

Таким образом, определение функции в пределах замкнутого выражения может повлиять на продолжительность существования замкнутого выражения; это *также* позволяет нам получить доступ к сущностям, к которым у нас обычно не было доступа. Рассмотрим пример.

```
let f;                           // неопределенная функция
{
    let o = { note: 'Безопасно' };
```

```
    f = function() {
        return o;
    }
}
let oRef = f();
oRef.note = "Все же не совсем безопасно!";
```

Обычно нечто вне области видимости строго недоступно. Функции являются исключением, они предоставляют нам окна в области видимости, которые в противном случае были бы недоступны. Мы убедимся в важности этого в следующих главах.

Немедленно вызываемые функциональные выражения

В главе 6 мы рассматривали функциональные выражения. Функциональные выражения позволяют создавать *немедленно вызываемые функциональные выражения* (Immediately Invoked Function Expression — IIFE). IIFE объявляет функцию, а затем немедленно ее запускает. Теперь, имея понятие об областях видимости и замкнутых выражениях, можно обсудить, зачем они могли бы нам понадобиться. IIFE выглядит следующим образом.

```
(function() {
    // Это тело IIFE
})();
```

Мы создаем анонимную функцию, используя функциональное выражение, а затем немедленно ее вызываем. Преимущество IIFE в том, что все в ней имеет собственную область видимости, а также, поскольку это функция, она может передать нечто из области видимости.

```
const message = (function() {
    const secret = "Здесь указан пароль!";
    return 'Пароль имеет длину ${secret.length} символов.';
})();
console.log(message);
```

Переменная `secret` защищена в области видимости IIFE, к ней нельзя обратиться извне. Вы можете возвратить из IIFE все, что хотите, и весьма часто возвращают массивы, объекты и функции. Давайте рассмотрим функцию, которая способна сообщить о количестве вызовов и в которую нельзя вмешаться.

```
const f = (function() {
    let count = 0;
    return function() {
        return 'Меня вызывали ${++count} раз(а).';
}
```

```
})();
f(); // " раз(а)."
f(); // "Меня вызывали 2 раз(а)."
//...
```

Поскольку переменная `count` надежно защищена в IIFE, нет никакого способа изменить ее из вне: у функции `f` всегда будет точный подсчет количества раз, когда она была вызвана.

Хотя использование переменных из области видимости блока в ES6 несколько снизило потребность в IIFE, последнее все еще весьма популярно и полезно, когда нужно создать замкнутое выражение и возвратить нечто из него.

Область видимости функции и механизм подъема объявлений

До введения в ES6 ключевого слова `let` переменные объявлялись с ключевым словом `var` и имели всю *область видимости функции* (function scope) (глобальные переменные, объявленные с ключевым словом `var` вне функции, имеют то же поведение).

Когда вы объявляете переменную с ключевым словом `let`, она не будет существовать в коде до момента ее объявления. Когда вы объявляете переменную с ключевым словом `var`, она будет доступна *повсюду в текущей области видимости*... даже перед ее оператором объявления. Прежде чем мы рассмотрим пример, запомните, что есть различие между переменной, которая не объявлена, и переменной, которой присвоено значение `undefined`. Необъявленные переменные приводят к ошибке, тогда как переменные, которые существуют, но имеют значение `undefined`, — нет.

```
let var1;
let var2 = undefined;
var1;         // undefined
var2;         // undefined
undefinedVar; // ReferenceError: undefinedVar  не определена
```

При использовании ключевого слова `let` вы получите ошибку, если попытаетесь обратиться к переменной до ее объявления.

```
x;         // ReferenceError: x не определена
let x = 3; // мы никогда не дойдем сюда - ошибка остановит выполнение программы
```

К переменным, объявленным с ключевым словом `var`, напротив, можно обращаться прежде, чем они будут объявлены.

```
x;         // undefined
var x = 3;
x;         // 3
```

Так что же здесь происходит? На первый взгляд, не имеет никакого смысла обращаться к переменной до ее объявления. Однако к переменным, объявленным с ключевым словом `var`, применяется механизм *подъема* (hoisting). JavaScript просматривает всю область видимости (функции или глобальную) и поднимает к ее вершине все переменные, объявленные с ключевым словом `var`. Важно понимать, что поднимаются только объявления, *а не присвоения*. Таким образом, JavaScript интерпретировал бы предыдущий пример так.

```
var x;      // поднято объявление (но не присвоение)
x;          // undefined
x = 3;
x;          // 3
```

Давайте рассмотрим более сложный пример наряду со способом, которым JavaScript его интерпретирует.

```
// что вы пишете

if(x !== 3) {
    console.log(y);
    var y = 5;
    if(y === 5) {
        var x = 3;
    }
    console.log(y);
}
if(x === 3) {
    console.log(y);
}
```

```
// как JavaScript интерпретирует это
var x;
var y;
if(x !== 3) {
    console.log(y);
    y = 5;
    if(y === 5) {
        x = 3;
    }
    console.log(y);
}
if(x === 3) {
    console.log(y);
}
```

Я не утверждаю, что это хорошо написанный код на JavaScript. Не нужно использовать переменные прежде, чем вы их объявите. Это ведет к ошибкам и не имеет никакого практического смысла. Но в данном примере действительно поясняется, как работает механизм подъема.

Еще один аспект переменных, объявленных с ключевым словом `var`, — движок JavaScript не обращает внимание на их повторное объявление.

```
// что вы пишете

var x = 3;
if(x === 3) {
    var x = 2;
    console.log(x);
}
console.log(x);
```

```
// как JavaScript интерпретирует это
var x;
x = 3;
if(x === 3) {
    x = 2;
    console.log(x);
}
console.log(x);
```

Этот пример должен прояснить, что (в пределах той же функции или глобальной области видимости) ключевое слово `var` не может использоваться для создания новых

переменных. Поэтому в данном случае маскировки переменных не происходит так, как это делается с помощью ключевого слова `let`. В данном примере есть только одна переменная `x`, даже при том что в блоке есть второе определение с ключевым словом `var`.

Это снова то, чего я не рекомендую делать во избежание возможных ошибок. Случайный читатель (особенно знакомый с другими языками программирования) может посмотреть на этот пример и резонно предположить, что автор намеревался создать новую переменную `x` в области видимости блока, созданного оператором `if`, чего на самом деле не было.

Если вы задались вопросом "Почему ключевое слово `var` позволяет делать такие запутанные и бесполезные вещи?", то вы теперь понимаете, почему появилось ключевое слово `let`. Конечно, вы можете и дальше использовать ключевое слово `var` ответственно и однозначно, но при этом случайно можно очень легко написать код, который окажется двусмысленным и неясным. В спецификации ES6 не могли просто "исправить" ключевое слово `var`, поскольку это нарушило бы работоспособность существующего кода; поэтому было введено ключевое слово `let`.

Я не могу придумать пример использования ключевого слова `var`, который нельзя было бы переписать лучше или яснее с использованием ключевого слова `let`. Другими словами, ключевое слово `var` не имеет преимуществ перед ключевым словом `let`, и многие в сообществе JavaScript (включая меня самого) полагают, что ключевое слово `let` в конечном счете полностью заменит ключевое слово `var` (и даже возможно, что определения с ключевым словом `var` в конечном счете устареют).

Итак, зачем же нужно изучать ключевое слово `var` и механизм подъема? По двум причинам. Во-первых, спецификация ES6 не будет общепринята еще на протяжении некоторого времени, а значит, код придется транскомпилировать в ES5, и, конечно, существует много кода, написанного в ES5. Поэтому в течение некоторого времени еще будет важно понимать, как работает ключевое слово `var`. Во-вторых, объявления функции также поднимаются, что подводит нас к следующей теме.

Подъем функций

Подобно объявлениям переменных с использованием ключевого слова `var`, объявления функций поднимаются к началу их области видимости. Это позволяет вызывать функции прежде, чем они будут объявлены.

```
f();                    // выводит "f"
function f() {
    console.log('f');
}
```

Обратите внимание, что функциональные выражения, которые присваиваются переменным, *не поднимаются*. Они просто подчиняются правилам областей видимости для переменных, как показано ниже.

```
f();                    // TypeError: f - не функция
let f = function() {
    console.log('f');
}
```

Временная мертвая зона

Временная мертвая зона (Temporal Dead Zone — TDZ) — это образное название для интуитивно понятной концепции, согласно которой переменные, объявляемые с ключевым словом `let`, не существуют в коде до момента их объявления. Для переменной к временной мертвой зоне в пределах области видимости относится тот код, который предшествует ее объявлению.

По большей части это не должно вызывать недопонимания или проблем, но есть один аспект TDZ, который собьет с толку людей, знакомых с JavaScript до ES6.

Оператор `typeof` общепринят для определения, была ли переменная объявлена, и считается "безопасным" способом проверки ее существования. Таким образом, до появления ключевого слова `let` в пределах TDZ, это всегда безопасно срабатывало для любого идентификатора `x` и не заканчивалось ошибкой.

```
if(typeof x === "undefined") {
    console.log("x не существует или равен undefined");
} else {
    // безопасное обращение к x....
}
```

При объявлении переменных с ключевым словом `let` этот код больше нельзя считать безопасным. Например, следующий код закончится ошибкой.

```
if(typeof x === "undefined") {
    console.log("x не существует или равен undefined ");
} else {
    // безопасное обращение к x....
}
let x = 5;
```

Проверка определенности переменных с использованием `typeof` в ES6 будет менее необходима, поэтому на практике поведение оператора `typeof` во временной мертвой зоне не должно вызывать проблем.

Строгий режим

Синтаксис ES5 допускал *неявные глобальные переменные* (implicit global), которые были источником многих ошибок в программе. Короче говоря, если вы забывали объявить переменную с ключевым словом `var`, то JavaScript беззаботно подразумевал, что вы обращаетесь к глобальной переменной. Если до этого никакой такой

глобальной переменной не существовало, то она тут же создавалась! Можете представить себе проблемы, к которым это приводило.

По этой причине (и ряду других) в JavaScript была введена концепция *строгого режима* (strict mode), предотвращающего неявные глобальные переменные. Строгий режим включается с помощью строкового литерала `"use strict"` (здесь вы можете использовать одиночные или двойные кавычки), расположенного в отдельной строке, перед любым другим кодом. Если сделать это в глобальной области видимости, весь сценарий будет выполняться в строгом режиме, а если сделать это в функции, то в строгом режиме будет выполняться только функция.

Поскольку строгий режим относится ко всему сценарию, если перейти к нему в глобальной области видимости, то могут возникнуть проблемы. На многих современных веб-сайтах используются вместе различные сценарии, написанные разными людьми. Поэтому переход в строгий режим в глобальной области видимости в одном из таких сценариев переводит в строгий режим их все. Хотя было бы, конечно, хорошо, чтобы все сценарии работали правильно в строгом режиме, но это далеко не так. Значит, обычно нецелесообразно использовать строгий режим в глобальной области видимости. Если вы не хотите включать строгий режим в каждой функции по отдельности (и кто бы это захотел делать?), можете заключить весь свой код в одну немедленно выполняемую функцию (больше об этом мы узнаем в главе 13).

```
(function() {
'use strict';

    // весь ваш код будет здесь..., он
    // выполняется в строгом режиме, но
    // строгий режим не будет затрагивать
    // никаких других сценариев, запущенных
    // вместе с этим
})();
```

Строгий режим считается хорошей вещью, и я рекомендую его использовать. Если вы будете использовать анализатор (что обязательно нужно делать!), то это предотвратит большинство распространенных проблем, но страховка никогда не помешает!

Чтобы узнать больше о строгом режиме, читайте соответствующую статью в библиотеке MDN.

Заключение

Знать области видимости важно для изучения любого языка программирования. Введение ключевого слова `let` приводит JavaScript в соответствие с большинством других современных языков. Хотя JavaScript — не первый язык, который поддерживает замкнутые выражения, это один из первых *популярных* (не академических) языков, сделавших это. В сообществе JavaScript замкнутые выражения используются для пущего эффекта, но это важная часть современной разработки JavaScript.

Массивы и их обработка

Массивы — одно из моих самых любимых средств языка JavaScript. Очень многие задачи программирования подразумевают манипулирование коллекциями данных, и свободное владение методами обработки массивов в JavaScript существенно облегчает это. Изучение этих методов является также отличным способом достижения следующего уровня мастерства в JavaScript.

Обзор массивов

Прежде чем продолжить, давайте вспомним об основах массивов. Массивы (в отличие от объектов) имеют упорядоченный характер, а числовые индексы их элементов отсчитываются от нуля. Массивы в JavaScript могут быть *негомогенными*, т.е. их элементы не обязаны иметь одинаковый тип (из этого следует, что элементами массивов могут быть другие массивы или объекты). Литеральные массивы создаются с помощью квадратных скобок, и те же квадратные скобки используются для доступа к элементам массива по индексу. Каждый массив имеет свойство `length`, указывающее количество элементов в массиве. Присвоение значения по индексу, превосходящему размер массива, автоматически приводит к увеличению массива, а неиспользуемые индексы получают значение `undefined`. Для создания массива можно также использовать конструктор `Array`, хотя это редко необходимо. Удостоверьтесь, что все нижеследующее вам понятно, прежде чем переходить далее.

```
// литеральные массивы
const arr1 = [1, 2, 3];                          // массив чисел
const arr2 = ["one", 2, "three"];                // негомогенный массив
const arr3 = [[1, 2, 3], ["one", 2, "three"]]; // массив, содержащий
                                                 // массивы
const arr4 = [                                   // негомогенный массив
    { name: "Fred", type: "object", luckyNumbers = [5, 7, 13] },
     [
          { name: "Susan", type: "object" },
          { name: "Anthony", type: "object" },
     ],
     1,
```

```
    function() { return "в элементе массива может также находиться и функция";
},
    "three",
];

// доступ к элементам
arr1[0];                        // 1
arr1[2];                        // 3
arr3[1];                        // ["one", 2, "three"]
arr4[1][0];                     // { name: "Susan", type: "object" }

// длина массива
arr1.length;                    // 3
arr4.length;                    // 5
arr4[1].length;                 // 2

// увеличение размера массива
arr1[4] = 5;
arr1;                           // [1, 2, 3, undefined, 5]
arr1.length;                    // 5

// при доступе (не присвоении) по индексу, большему, чем есть
// в массиве, размер массива *не* изменяется
arr2[10];                       // undefined
arr2.length;                    // 3

// Конструктор Array (используется редко)
const arr5 = new Array();         // пустой массив
const arr6 = new Array(1, 2, 3); // [1, 2, 3]
const arr7 = new Array(2);        // массив длиной 2 (все
                                  // элементы undefined)
const arr8 = new Array("2");      // ["2"]
```

Манипулирование содержимым массива

Прежде чем продолжить, рассмотрим весьма полезные методы манипулирования массивами. Один из аспектов обработки массивов, к сожалению, невразумительный, относится к различию между методами, изменяющими сам массив "по месту", и методами, возвращающими новый массив. Никакого соглашения по этому поводу нет, и это только один из нюансов, которые вам придется запомнить (например, метод `push` изменяет сам массив, а `concat` — возвращает новый массив).

В некоторых языках, таких как Ruby, есть соглашения, которые облегчают определение, модифицирует ли метод нечто по месту или возвращает копию. Например, в Ruby, если у вас есть строка `str` и вы вызываете метод `str.downcase`, он возвратит литерал в нижнем

регистре, но сама str останется неизменной. С другой стороны, если вы вызываете str.downcase!, то это изменит саму строку str. Тот факт, что стандартные библиотеки JavaScript не предоставляют информации о том, какие методы возвращают копию, а какие модифицируют источник, по моему мнению, является одним из недостатков языка, требующего ненужного запоминания.

Добавление отдельных элементов в начало или конец и их удаление

Обращаясь к *началу* массива, мы обращаемся к его первому элементу (элементу 0). Аналогично *конец* массива является элементом с наибольшим индексом (точнее элементом arr.length-1 массива arr). Методы push и pop добавляют элементы в конец массива (по месту) и удаляют их соответственно. Методы shift и unshift добавляют элементы в начало массива (по месту) и удаляют их соответственно.

Названия этих методов происходят от терминов из информатики. *Помещение* (push) и *извлечение* (pop) — это действия со *стеком* (stack), в котором первым извлекается элемент, добавленный последним. Методы shift и unshift обрабатывают массив как *очередь* (queue), в которой первым извлекается элемент, добавленный в очередь первым.

Методы push и unshift возвращают новую длину массива после добавления нового элемента, а pop и shift возвращают удаленный элемент. Вот примеры этих методов в действии.

```
const arr = ["b", "c", "d"];
arr.push("e");          // возвращает 4; теперь arr ["b", "c", "d", "e"]
arr.pop();              // возвращает "e"; теперь arr ["b", "c", "d"]
arr.unshift("a");       // возвращает 4; теперь arr ["a", "b", "c", "d"]
arr.shift();            // возвращает "a"; теперь arr ["b", "c", "d"]
```

Добавление нескольких элементов в конец

Метод concat добавляет в массив несколько элементов и возвращает его копию. Если передать методу concat массивы, он разделит их и добавит их элементы в исходный массив. Рассмотрим примеры.

```
const arr = [1, 2, 3];
arr.concat(4, 5, 6);      // возвращает [1, 2, 3, 4, 5, 6]; arr неизменен
arr.concat([4, 5, 6]);    // возвращает [1, 2, 3, 4, 5, 6]; arr неизменен
arr.concat([4, 5], 6);    // возвращает [1, 2, 3, 4, 5, 6]; arr неизменен
arr.concat([4, [5, 6]]); // возвращает [1, 2, 3, 4, [5, 6]]; arr неизменен
```

Обратите внимание, что `concat` разделяет массивы, предоставленные только непосредственно; он не разделяет массивы в этих массивах.

Получение подмассива

Если вы хотите получить подмассив из массива, используйте метод `slice`, которому можно передать два аргумента. Первый аргумент — индекс начала подмассива, а второй — индекс его конца (не включая указанный элемент). Если пропустить конечный аргумент, возвратятся все элементы до конца массива. Этот метод позволяет использовать отрицательные индексы для ссылки на элементы относительно конца массива, что весьма удобно. Рассмотрим примеры.

```
const arr = [1, 2, 3, 4, 5];
arr.slice(3);               // возвращает [4, 5]; arr неизменен
arr.slice(2, 4);            // возвращает [3, 4]; arr неизменен
arr.slice(-2);              // возвращает [4, 5]; arr неизменен
arr.slice(1, -2);           // возвращает [2, 3]; arr неизменен
arr.slice(-2, -1);          // возвращает [4]; arr неизменен
```

Добавление и удаление элементов в любой позиции

Метод `splice` позволяет изменять массив по месту, добавляя и/или удаляя элементы из любого индекса. Первый аргумент — индекс, с которого должно начинаться изменение; второй аргумент — количество удаляемых элементов (если вы не хотите удалять элементы, используйте `0`), а остальные аргументы — это добавляемые элементы. Рассмотрим примеры.

```
const arr = [1, 5, 7];
arr.splice(1, 0, 2, 3, 4);  // возвращает []; теперь arr [1, 2, 3, 4, 5, 7]
arr.splice(5, 0, 6);        // возвращает []; теперь arr [1, 2, 3, 4, 5, 6, 7]
arr.splice(1, 2);           // возвращает [2, 3]; теперь arr [1, 4, 5, 6, 7]
arr.splice(2, 1, 'a', 'b'); // возвращает [5]; теперь arr [1, 4, 'a', 'b', 6, 7]
```

Копирование и вставка в пределах массива

Спецификация ES6 представляет новый метод, `copyWithin`, получающий последовательность элементов из массива, и копирующий по месту, в другую часть массива, переписывая любые находящиеся там элементы. Первый аргумент — откуда копировать, второй аргумент — куда копировать, а заключительный (необязательный) аргумент — где прекратить копирование. Как и в методе `slice`, вы можете использовать отрицательные числа для индексов начала и завершения; они рассчитываются от конца массива. Рассмотрим примеры.

```
const arr = [1, 2, 3, 4];
arr.copyWithin(1, 2);       // теперь arr [1, 3, 4, 4]
```

```
arr.copyWithin(2, 0, 2);   // теперь arr [1, 3, 1, 3]
arr.copyWithin(0, -3, -1); // теперь arr [3, 1, 1, 3]
```

Заполнение массива заданным значением

Спецификация ES6 вводит новый метод, `fill`, который позволяет задать любое количество элементов с фиксированным значением (по месту). Он особенно полезен, когда используется вместе с конструктором `Array` (который позволяет определить начальный размер массива). Вы можете произвольно задать начальный и конечный индексы, если хотите заполнить только часть массива (отрицательные индексы работают как обычно). Рассмотрим примеры.

```
const arr = new Array(5).fill(1); // arr инициализируется [1, 1, 1, 1, 1]
arr.fill("a");                    // теперь arr ["a", "a", "a", "a", "a"]
arr.fill("b", 1);                 // теперь arr ["a", "b", "b", "b", "b"]
arr.fill("c", 2, 4);              // теперь arr ["a", "b", "c", "c", "b"]
arr.fill(5.5, -4);                // теперь arr ["a", 5.5, 5.5, 5.5, 5.5]
arr.fill(0, -3, -1);              // теперь arr ["a", 5.5, 0, 0, 5.5]
```

Обращение и сортировка массивов

Метод `reverse` прост, он изменяет порядок элементов массива на обратный (по месту).

```
const arr = [1, 2, 3, 4, 5];
arr.reverse();     // теперь arr [5, 4, 3, 2, 1]
```

Метод `sort` сортирует массив (по месту).

```
const arr = [5, 3, 2, 4, 1];

arr.sort();        // теперь arr [1, 2, 3, 4, 5]
```

Метод `sort` позволяет также определить *функцию сортировки* (sort function), которая может оказаться весьма удобной. Например, для сортировки объектов нет однозначного способа.

```
const arr = [{ name: "Suzanne" }, { name: "Jim" },
             { name: "Trevor" },  { name: "Amanda" }];
arr.sort();                                  // arr неизменен
arr.sort((a, b) => a.name > b.name);         // arr отсортирован в
                                             // алфавитном порядке по
                                             // свойству name
arr.sort((a, b) => a.name[1] < b.name[1]); // arr отсортирован в обратном
                                             // алфавитному порядке по
                                             // второму символу в
                                             // свойстве name
```

В данном примере мы возвращаем логическое значение. Но метод `sort` понимает также и число в виде возвращаемого значения. Если вы возвратите `0`, то метод `sort` будет полагать, что эти два элемента "равны", и оставит порядок неизменным. Это позволило бы нам, например, сортировать в алфавитном порядке, за исключением слов, начинающихся с символа *k*. При этом все было бы отсортировано в алфавитном порядке, со всеми словами *k*, расположенными после всех слов *j* и перед всеми словами *l*, но слова *k* будут в их исходном порядке (т.е. несортированными).

Поиск в массиве

Если вы хотите найти что-то в массиве, у вас есть несколько возможностей. Начнем со скромного метода `indexOf`, который был доступен в JavaScript довольно давно. Метод `indexOf` просто возвращает индекс первого найденного элемента, строго равного искомому (есть соответствующий метод `lastIndexOf`, осуществляющий поиск в обратном направлении и возвращающий последний индекс, который соответствует искомому). Чтобы выполнять поиск только в части массива, можно определить необязательный индекс начала. Если `indexOf` (или `lastIndexOf`) возвращает `-1`, это означает, что соответствие не найдено.

```
const o = { name: "Jerry" };
const arr = [1, 5, "a", o, true, 5, [1, 2], "9"];
arr.indexOf(5);                  // возвращает 1
arr.lastIndexOf(5);              // возвращает 5
arr.indexOf("a");                // возвращает 2
arr.lastIndexOf("a");            // возвращает 2
arr.indexOf({ name: "Jerry" }); // возвращает -1
arr.indexOf(o);                  // возвращает 3
arr.indexOf([1, 2]);             // возвращает -1
arr.indexOf("9");                // возвращает 7
arr.indexOf(9);                  // возвращает -1
arr.indexOf("a", 5);             // возвращает -1
arr.indexOf(5, 5);               // возвращает 5
arr.lastIndexOf(5, 4);           // возвращает 1
arr.lastIndexOf(true, 3);        // возвращает -1
```

Далее, метод `findIndex` подобен методу `indexOf` в том, что возвращает индекс (или `-1` при отсутствии соответствия), но более гибко. Он позволяет задать функцию, которая определяет, является ли элемент соответствующим (`findIndex` не может начать работу с произвольного индекса и не имеет аналога `lastIndexOf`).

```
const arr = [{ id: 5, name: "Judith" }, { id: 7, name: "Francis" }];
arr.findIndex(o => o.id === 5);            // возвращает 0
arr.findIndex(o => o.name === "Francis"); // возвращает 1
```

```
arr.findIndex(o => o === 3);              // возвращает -1
arr.findIndex(o => o.id === 17);          // возвращает -1
```

Методы `find` и `findIndex` применяются при поиске индекса элемента. Но что если индекс элемента не интересен, а нужен только сам элемент? Метод `find` похож на `findIndex` тем, что позволяет определять функцию для поиска, но возвращает сам элемент, а не индекс (или `null`, если элемент не был найден).

```
const arr = [{ id: 5, name: "Judith" }, { id: 7, name: "Francis" }];
arr.find(o => o.id === 5); // возвращает объект { id: 5, name: "Judith" }
arr.find(o => o.id === 2); // возвращает null
```

Функции, которые вы передаете методам `find` и `findIndex`, получают, кроме каждого элемента в их первом аргументе, также индекс текущего элемента и весь сам массив в качестве аргументов. Это позволяет осуществлять, например, поиск квадратов чисел, соответствующих определенным индексам.

```
const arr = [1, 17, 16, 5, 4, 16, 10, 3, 49];
arr.find((x, i) => i > 2 && Number.isInteger(Math.sqrt(x))); // возвращает 4
```

Методы `find` и `findIndex` позволяют также использовать переменную `this` во время вызова функции. Это может быть очень удобно, если вам нужно вызвать функцию, как будто она является методом объекта. Рассмотрим следующие эквивалентные методики поиска объекта `Person` по идентификатору.

```
class Person {
    constructor(name) {
        this.name = name;
        this.id = Person.nextId++;
    }
}
Person.nextId = 0;
const jamie = new Person("Jamie"),
     juliet = new Person("Juliet"),
      peter = new Person("Peter"),
        jay = new Person("Jay");
const arr = [jamie, juliet, peter, jay];

// возможность 1: прямое сравнение идентификатора:
arr.find(p => p.id === juliet.id);        // возвращает объект juliet

// возможность 2: использование аргумента "this":
arr.find(p => p.id === this.id, juliet); // возвращает объект juliet
```

Вы, вероятно, сейчас найдете не много поводов для определения значения `this` в функциях `find` и `findIndex`, но впоследствии вы увидите, где эта методика весьма полезна.

Подобно тому, как нас не всегда заботит индекс элемента в пределах массива, сам элемент нас тоже не всегда интересует: иногда мы просто хотим знать, есть он или нет. Очевидно, мы можем использовать одну из приведенных выше функций и выяснить, возвращает ли она `-1` или `null`, но в JavaScript есть для этого два метода: `some` и `every`.

Метод `some` возвращает `true`, если находит элемент, который соответствует критерию (это все, что нужно, дальнейший поиск сразу прекращается), и `false` в противном случае. Вот пример.

```
const arr = [5, 7, 12, 15, 17];
arr.some(x => x%2===0);                       // true; 12 четно
arr.some(x => Number.isInteger(Math.sqrt(x))); // false; нет квадратов
```

Метод `every` возвращает `true`, если каждый элемент в массиве удовлетворяет критерию, и `false` в противном случае. Он прекращает поиск и возвращает `false`, как только найдет элемент, не соответствующий критерию; в противном случае он должен будет просмотреть весь массив.

```
const arr = [4, 6, 16, 36];
arr.every(x => x%2===0);                      // true; нет нечетных чисел
arr.every(x => Number.isInteger(Math.sqrt(x))); // false; 6 - не квадрат
```

Как и все методы в этой главе, которым передается проверочная функция, `some` и `every` имеют второй параметр, который позволяет вам определить значение `this` при вызове функции.

Фундаментальные операции над массивом: `map` и `filter`

Из всех операций над массивом `map` и `filter` вы найдете самыми полезными. Просто удивительно, чего можно достичь с помощью этих двух методов.

Метод `map` *преобразует* элементы в массиве. Во что? Это вам решать. У вас есть объекты, которые содержат числа, а вам нужны именно сами числа? Легко! Ваш массив содержит функции, а нужны возвращаемые ими значения? Легко! *Всякий раз, когда массив находится в одном формате, а необходим другой, используйте метод* `map`. Методы `map` и `filter` возвращают копии и не изменяют исходный массив. Давайте рассмотрим несколько примеров.

```
const cart = [ { name: "Widget", price: 9.95 }, { name: "Gadget", price: 22.95
}];
const names = cart.map(x => x.name);           // ["Widget", "Gadget"]
const prices = cart.map(x => x.price);         // [9.95, 22.95]
const discountPrices = prices.map(x => x*0.8); // [7.96, 18.36]
const lcNames = names.map(String.toLowerCase); // ["widget", "gadget"]
```

Вы можете задаваться вопросом "Как работает lcNames? Этот случай выглядит не так, как другие". Все обсуждаемые здесь методы, которым передаются функции, включая map, не заботятся о том, в каком виде им передается эта функция. В случаях names, prices и discountPrices мы создаем собственную функцию (используя стрелочную нотацию). Для lcNames мы используем функцию, которая уже существует, String.toLowerCase. Этой функции передается один строковый аргумент, а она возвращает строку в нижнем регистре. Мы легко могли бы написать names.map(x => x.toLowerCase()), но важно понимать, что функция — это функция, независимо от того, какую форму она принимает.

Передаваемая в качестве параметра функция вызывается для каждого элемента. Ей передаются три аргумента: сам элемент, его индекс в массиве и сам массив (который редко полезен). Рассмотрим пример, в котором имеются наши товары и соответствующие цены в двух отдельных массивах, а мы хотим объединить их.

```
const items = ["Widget", "Gadget"];
const prices = [9.95, 22.95];
const cart = items.map((x, i) => ({ name: x, price: prices[i]}));
// cart: [{ name: "Widget", price: 9.95 }, { name: "Gadget", price: 22.95 }]
```

Этот пример немного сложнее, но в нем демонстрируется вся мощь функции map. Здесь, мы используем не только сам элемент (x), но и его индекс (i). Индекс необходим потому, что мы хотим соотнести элементы в массиве items с элементами в массиве prices согласно их индексу. Здесь метод map преобразует массив строк в массив объектов, извлекая информацию из отдельных массивов. (Обратите внимание, что мы должны заключить объект в круглые скобки; без круглых скобок стрелочная нотация примет фигурные скобки за обозначение блока.)

Метод filter, как и подразумевает его имя, предназначен для удаления всего нежелательного из массива. Как и map, после удаления элементов он возвращает новый массив. Какие элементы удаляются? Это снова полностью ваше дело. Если вы догадались, что для определения удаляемых элементов мы предоставляем функцию, то вы уловили суть. Давайте рассмотрим несколько примеров.

```
// создать колоду игральных карт
const cards = [];
for(let suit of ['H', 'C', 'D', 'S']) // червы, трефы, бубны, пики
    for(let value=1; value<=13; value++)
        cards.push({ suit, value });

// получить все карты со значением 2:
cards.filter(c => c.value === 2); // [
                                  // { suit: 'H', value: 2 },
                                  // { suit: 'C', value: 2 },
                                  // { suit: 'D', value: 2 },
                                  // { suit: 'S', value: 2 }
```

```
// ]

// (далее для краткости мы выводим только длину)

// получить все бубны:
cards.filter(c => c.suit === 'D');                    // длина: 13

// получить все фигуры
cards.filter(c => c.value > 10);                      // длина: 12

// получить все червовые фигуры
cards.filter(c => c.value > 10 && c.suit === 'H'); // длина: 3
```

Полагаю, вы начали понимать, как методы `map` и `filter` могут быть объединены для получения интересного эффекта. Скажем, например, мы хотим создать краткое строковое представление карт в нашей колоде. Мы будем использовать символы Юникода для мастей и буквы "A", "J", "Q" и "K" для обозначения туза и фигур. Поскольку создающая функция довольно длинна, мы создадим ее отдельно и не будем пытаться использовать анонимную функцию.

```
function cardToString(c) {
    const suits = { 'H': '\u2665', 'C': '\u2663', 'D': '\u2666', 'S': '\u2660'
};
    const values = { 1: 'A', 11: 'J', 12: 'Q', 13: 'K' };
    // создание массива значений при каждом вызове функции cardToString
    // не очень эффективно; попробуйте найти лучшее решение
    for(let i=2; i<=10; i++) values[i] = i;
    return values[c.value] + suits[c.suit];
}

// получить все карты со значением 2:
cards.filter(c => c.value === 2)
    .map(cardToString);      // [ «2♥», «2♣», «2♦», «2♣» ]

// получить все червовые фигуры
cards.filter(c => c.value > 10 && c.suit === 'H')
    .map(cardToString);      // [ «J♥», «Q♥», «K♥» ]
```

Магия массивов: метод `reduce`

Из всех методов массивов мой любимый — `reduce`. В то время как `map` преобразует каждый элемент в массиве, метод `reduce` преобразует *весь массив*. Он называется `reduce` потому, что зачастую используется для сведения (reduce) массива к единому значению. Например, суммирование чисел, хранящихся в массиве, или вычисление их среднего являются способами свести массив к единому значению. Однако фактически

результатом сведения к единому значению может быть объект или другой массив — метод reduce способен воспроизвести возможности функций map и filter (и если на то пошло, любой другой рассмотренной здесь функции массива).

Метод reduce, подобно map и filter, позволяет предоставить функцию, которая контролирует результат. Прежде мы уже имели дело с функциями *обратного вызова* (callback), первый переданный им элемент всегда является текущим элементом массива. Однако первое значение функции reduce — *аккумулятор* (accumulator), в который сводится массив. Остальная часть аргументов вполне ожидаема: текущий элемент массива, текущий индекс и сам массив.

Помимо функции обратного вызова, методу reduce передается (необязательно) начальное значение для аккумулятора. Давайте рассмотрим простой пример — суммирование чисел в массиве.

```
const arr = [5, 7, 2, 4];
const sum = arr.reduce((a, x) => a += x, 0);
```

Передаваемая в reduce функция получает два параметра: аккумулятор (a) и текущий элемент массива (x). В этом примере аккумулятор изначально содержит значение 0. Поскольку это наш первый опыт с reduce, рассмотрим все этапы, которые проходит JavaScript, чтобы лучше понять, как это работает.

1. Для первого элемента массива (5) вызывается (анонимная) функция. Первоначально a имеет значение 0, а x — значение 5. Функция возвращает сумму a и x (5), что становится значением a на следующем этапе.

2. Функция вызывается для второго элемента массива (7). Теперь a имеет значение 5 (переданное с предыдущего этапа), а x имеет значение 7. Функция возвращает сумму a и x (12), которая становится значением a на следующем этапе.

3. Функция вызывается для третьего элемента массива (2). Теперь a имеет значение 12, а x — значение 2. Функция возвращает сумму a и x (14).

4. Функция вызывается для четвертого, и последнего, элемента массива (4). Теперь a имеет значение 14, а x — значение 4. Функция возвращает сумму a и x (18), которая и будет возвращаемым значением функции reduce (которое затем присваивается константе sum).

Проницательный читатель уже мог бы понять, что в этом очень простом примере мы даже не должны присваивать значение a; важнее всего то, что возвращается из функции (помните, что стрелочная нотация не требует явного оператора return), таким образом, мы можем просто возвращать a + x. Однако в более сложных примерах нам может понадобиться сделать с аккумулятором нечто большее. Таким образом, модификация аккумулятора в функции — это хорошая привычка.

Прежде чем мы перейдем к более интересным случаям применения метода reduce, давайте рассмотрим, что будет, если аккумулятору не присваивается начальное значение, т.е. он равен undefined. Тогда метод reduce считает первый элемент массива в качестве начального значения и начинает вызывать функцию со вторым элементом. Давайте вернемся к нашему примеру, но опустим начальное значение.

```
const arr = [5, 7, 2, 4];
const sum = arr.reduce((a, x) => a += x);
```

1. Для *второго* элемента массива (7) вызывается (анонимная) функция. У a теперь начальное значение 5 (первый элемент массива), а x содержит значение 7. Функция возвращает сумму a и x (12), что становится значением a на следующем этапе.
2. Функция вызывается для третьего элемента массива (2). Теперь a имеет начальное значение 12, а x — значение 2. Функция возвращает сумму a и x (14).
3. Функция вызывается для четвертого, и последнего, элемента массива (4). Теперь a имеет значение 14, а x — значение 4. Функция возвращает сумму a и x (18), которая и становится возвращаемым значением reduce (оно затем присваивается константе sum).

Как можно заметить, здесь на один этап меньше, но результат то же. В этом примере (и в любом другом случае, когда первый элемент может служить начальным значением аккумулятора) мы можем извлечь пользу, исключив начальное значение.

Обычно для метода reduce в качестве аккумулятора используется значение базового типа (число или строка), но использование для аккумулятора объекта — это очень мощный подход (о котором часто забывают). Например, если у вас есть массив строк и вы хотите сгруппировать строки в упорядоченные по алфавиту массивы (слова на А, слова на Б и т.д.), можете использовать объект.

```
const words = ["Beachball", "Rodeo", "Angel",
    "Aardvark", "Xylophone", "November", "Chocolate",
    "Papaya", "Uniform", "Joker", "Clover", "Bali"];
const alphabetical = words.reduce((a, x) => {
    if(!a[x[0]]) a[x[0]] = [];
    a[x[0]].push(x);
    return a; }, {});
```

Этот пример немного сложнее, но принцип тот же. Для каждого элемента в массиве функция проверяет аккумулятор на наличие у него свойства для первой буквы в слове; если его нет, она добавляет пустой массив (когда она встречает "Beachball" и никакого свойства a.B нет, она создает для него пустой массив). Затем она добавляет слово в соответствующий массив (который, возможно, был только что создан), и наконец аккумулятор (a) возвращается (помните, что

значение, которое вы *возвращаете*, используется как аккумулятор для следующего элемента в массиве).

Другой пример — вычислительная статистика. Давайте, например, вычислим среднее и дисперсию для набора данных.

```
const data = [3.3, 5, 7.2, 12, 4, 6, 10.3];
// Алгоритм Дональда Кнута для вычисления дисперсии: Искусство
// программирования, том 2. Получисленные алгоритмы, 3-е изд. 2000 год
const stats = data.reduce((a, x) => {
        a.N++;
        let delta = x - a.mean;
        a.mean += delta/a.N;
        a.M2 += delta*(x - a.mean);
        return a;
    }, { N: 0, mean: 0, M2: 0 });
if(stats.N > 2) {
    stats.variance = stats.M2 / (stats.N - 1);
    stats.stdev = Math.sqrt(stats.variance);
}
```

И снова мы используем объект как аккумулятор, поскольку необходимо несколько переменных (в частности — `mean` и `M2`: при желании вместо `N` мы могли бы использовать индексный аргумент минус один).

Давайте рассмотрим еще один пример, демонстрирующий гибкость метода `reduce` при использовании аккумулятора, тип которого мы еще не применяли, — строки.

```
const words = ["Beachball", "Rodeo", "Angel",
    "Aardvark", "Xylophone", "November", "Chocolate",
    "Papaya", "Uniform", "Joker", "Clover", "Bali"];
const longWords = words.reduce((a, w) => w.length>6 ? a+" "+w : a, "").trim();
// longWords: "Beachball Aardvark Xylophone November Chocolate Uniform"
```

Здесь мы используем строковый аккумулятор для получения единой строки, содержащей все слова, которые состоят более чем из шести символов. В качестве самостоятельного упражнения попробуйте переписать его, используя вместо `reduce` методы `filter` и `join` (строковый метод). (Начните с ответа на вопрос "Почему необходимо вызвать `trim` после `reduce`?")

Надеюсь, вам понравилась мощь метода `reduce`. Из всех методов обработки массива этот является наиболее универсальным и мощным.

Методы массива и удаленные или еще не определенные элементы

Зачастую недопонимание поведения методов массива ведет к неправильным предположениям относительно обработки ими элементов, которые были удалены

или еще не были определены. Методы `map`, `filter` и `reduce` *не вызывают функцию для элементов, которые никогда не присваивались или были удалены.* Например, до ES6, если бы вы попытались хитро инициализировать массив таким способом, то были бы разочарованы.

```
const arr = Array(10).map(function(x) { return 5 });
```

Массив `arr` был бы массивом с 10 элементами, но все они содержали бы значение `undefined`. Точно так же, если вы удалите элемент из середины массива, а затем вызовете метод `map`, то получите массив с "дыркой".

```
const arr = [1, 2, 3, 4, 5];
delete arr[2];
arr.map(x => 0);    // [0, 0, <одно пустое место>, 0, 0]
```

На практике эта проблема возникает редко, поскольку обычно вы работаете с массивами, элементы которых заданы явно (а если вам специально понадобился промежуток в массиве, что бывает редко, вы не будете вызывать метод `delete` для всего массива), но знать об этом нужно.

Соединение строк

Довольно часто приходится объединять (строковые) значения элементов массива, используя некий разделитель. Функция `Array.prototype.join` получает один аргумент, разделитель (стандартно — запятая, если вы его не укажете), и возвращает строку с объединенными элементами (включая еще не определенные и удаленные элементы в виде пустых строк; значения `null` и `undefined` также становятся пустыми строками).

```
const arr = [1, null, "hello", "world", true, undefined];
delete arr[3];
arr.join();         // "1,,hello,,true,"
arr.join('');       // "1hellotrue"
arr.join(' -- ');   // "1 -- -- hello -- -- true --"
```

При грамотном использовании (совместно с объединением и конкатенацией строк) функция `Array.prototype.join` позволяет создать такие элементы, как списки HTML `<ul>`.

```
const attributes = ["Nimble", "Perceptive", "Generous"];
const html = '<ul><li>' + attributes.join('</li><li>') + '</li></ul>';
// html: "<ul><li>Nimble</li><li>Perceptive</li><li>Generous</li></ul>";
```

Будьте внимательны, не поступайте так с пустым массивом: вы получите один пустой элемент `<li>`!

Заключение

Встроенный класс JavaScript `Array` обладает большой мощью и гибкостью, но иногда может быть не до конца понятно, когда какой метод использовать. Возможности методов класса `Array` приведены в табл. 8.1–8.4.

Для методов `Array.prototype`, которым передается функция (`find`, `findIndex`, `some`, `every`, `map`, `filter` и `reduce`), предоставляемая функция получает аргументы, представленные в табл. 8.1, для каждого элемента в массиве.

Таблица 8.1. Аргументы функции массива (по порядку)

Метод	Описание
Только `reduce`	Аккумулятор (исходное значение или значение, возвращенное последним вызовом)
Все	Элемент (значение текущего элемента)
Все	Индекс текущего элемента
Все	Сам массив (редко полезен)

Всем методам `Array.prototype`, которым передается функция, можно также передать необязательное значение переменной `this`, позволяющее вызывать функции, как будто это метод.

Таблица 8.2. Манипулирование содержимым массива

Когда необходимо...	Используйте...	По месту или копия
Создать стек ("последним пришел, первым вышел" [LIFO])	`push` (возвращает новую длину), `pop`	По месту
Создать очередь ("первым пришел, первым вышел" [FIFO])	`unshift` (возвращает новую длину), `shift`	По месту
Добавить несколько элементов в конец	`concat`	Копия
Получить подмассив	`slice`	Копия
Добавить или удалить элементы в любой позиции	`splice`	По месту
Вырезка и замена в пределах массива	`copyWithin`	По месту
Заполнение массива	`fill`	По месту
Обращение массива	`reverse`	По месту
Сортировка массива	`sort` (передается функция для специальной сортировки)	По месту

Таблица 8.3. Поиск в массиве

Когда необходимо знать/найти...	Используйте...
Индекс элемента	`indexOf` (простые значения), `findIndex` (сложные значения)
Последний индекс элемента	`lastIndexOf` (простые значения)
Сам элемент	`find`
Есть ли в массиве элемент, удовлетворяющий некому критерию	`some`
Все ли элементы в массиве удовлетворяют некому критерию	`every`

Таблица 8.4. Преобразование массива

Когда необходимо...	Используйте...	По месту или копия
Преобразовать каждый элемент в массив	`map`	Копия
Удалить элементы из массива на основании неких критериев	`filter`	Копия
Преобразовать весь массив в другой тип данных	`reduce`	Копия
Преобразовать элементы в строки и объединить	`join`	Копия

Объекты и объектно-ориентированное программирование

Основы объектов JavaScript мы рассмотрели в главе 3, а теперь пришло время изучить их глубже.

Как и массивы, объекты в JavaScript — это *контейнеры*, которые называют *агрегатными* или комплексными *типами данных*. У объектов есть два основных отличия от массивов.

- Массивы содержат значения, индексированные в числовой форме; объекты содержат свойства, индексированные строкой или символом.
- Массивы упорядочены (элемент `arr[0]` всегда следует перед `arr[1]`); объекты не упорядочены (вы не можете гарантировать, что свойство `obj.a` расположено перед `obj.b`).

Эти различия носят довольно эзотерический (но важный) характер, поэтому давайте считать свойства тем, что делает объекты по настоящему особенными. *Свойство* (property) состоит из *ключа* (key) (строки или символа) и *значения* (value). Особенными объекты делает то, что вы можете обращаться к свойствам по их ключам.

Перебор свойств

Обычно, если нужно вывести содержимое некоторого контейнера (операция *перебора*), чаще всего используется массив, а не объект. Тем не менее объекты также являются контейнерами, и они обеспечивают перебор свойств; вам только нужно знать об особенностях и возможных сложностях.

Первое, что необходимо помнить о переборе свойств, — это то, что *порядок* не *гарантируется*. Проведя небольшой эксперимент, вы можете обнаружить, что свойства выводятся в том порядке, в котором они добавляются, и это справедливо для многих реализаций движка *почти всегда*. Тем не менее JavaScript не дает никаких гарантий этому, и изменение реализации движка в любой момент может

ликвидировать этот эффект. Поэтому не стоит полагаться на результат эксперимента и *никогда* не стоит подразумевать, что порядок свойств будет именно таков.

Будучи предупрежденными об этом, давайте теперь рассмотрим основные способы перебора свойств объекта.

Цикл `for...in`

Традиционным способом перебора свойств объекта является цикл `for...in`. Рассмотрим объект, у которого есть несколько строковых свойств и одно символьное свойство.

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

for(let prop in o) {
    if(!o.hasOwnProperty(prop)) continue;
    console.log('${prop}: ${o[prop]}');
}
```

Все кажется довольно простым... кроме, вероятно, вполне резонного вопроса "Что делает `hasOwnProperty`?" Он ликвидирует опасность, связанную с циклом `for...in`, которая станет ясна далее в этой главе. Речь идет об унаследованных свойствах. В данном примере мы это могли бы опустить и не придать значения. Но, перебирая свойства объектов других типов (особенно объектов, производных от других), вы можете обнаружить свойства, которых не ожидали. Поэтому я рекомендую выработать привычку использовать для проверки метод `hasOwnProperty`. Вы скоро узнаете, почему это так важно, а также научитесь определять, когда его можно безболезненно (или желательно) опустить.

Обратите внимание, что цикл `for...in` не выводит значения свойств с символьными ключами.

Несмотря на то что с помощью цикла `for...in` можно выполнить перебор элементов массива, обычно это считается плохой идеей. Для массивов я рекомендую использовать обычный цикл `for` или `forEach`.

Метод `Object.keys`

Метод `Object.keys` позволяет получить все перечислимые строковые свойства объекта в виде массива.

```
const SYM = Symbol();

const o = { a: 1, b: 2, c: 3, [SYM]: 4 };

Object.keys(o).forEach(prop => console.log('${prop}: ${o[prop]}'));
```

Этот пример приводит к тому же результату, что и цикл `for...in` (здесь даже не нужно выполнять проверку с помощью метода `hasOwnProperty`). Это весьма удобно, когда нужно собрать ключи свойств объекта в виде массива. Например, это облегчает вывод всех свойств объекта, которые начинаются с символа *x*.

```
const o = { apple: 1, xochitl: 2, balloon: 3, guitar: 4, xylophone: 5, };

Object.keys(o)
    .filter(prop => prop.match(/^x/))
    .forEach(prop => console.log('${prop}: ${o[prop]}'));
```

Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП, Object-Oriented Programming) — старая добрая парадигма в информатике. Некоторые из концепций, которые мы теперь знаем как ООП, появились еще в 1950-х годах, но только после появления языков программирования Simula 67 и Smalltalk они обрели форму ООП.

Фундаментальная идея проста и интуитивно понятна: *объект* — это логически связанная коллекция данных и функций. Она призвана отразить наше понимание естественного мира. *Автомобиль* — это объект, у которого есть данные (марка, модель, количество дверей, идентификатор транспортного средства (VIN) и т.д.), а также функции (ускорение, переключение передач, открывание дверей, включение фар и т.д.). Кроме того, ООП позволяет думать о вещах абстрактно (*автомобиль*) и конкретно (*определенный* автомобиль).

Прежде чем продолжать, давайте рассмотрим базовую лексику ООП. Термин *класс* (class) описывает обобщенную сущность (*автомобиль*), а *экземпляр* (instance) (или *экземпляр объекта* (object instance)) — определенную сущность (*конкретный* автомобиль, такой как "мой автомобиль"). Одна часть функций (ускорение) является *методами* (method). Другая часть функций, связанных с классом, но не относящихся к конкретному экземпляру, является *методами класса* (например, "создание нового VIN" могло бы быть методом класса: это не имеет отношения к конкретному новому автомобилю и, конечно, мы не ожидаем, что у конкретного автомобиля будет возможность или способность создать новый, законный VIN). Когда экземпляр создается, выполняется его *конструктор* (constructor). Конструктор инициализирует экземпляр объекта.

ООП предоставляет нам также среду для иерархической категоризации классов. Например, мог бы существовать более общий класс *транспортного средства*. У транспортного средства может быть характеристика *дальности* (дистанция, которую он может пойти без дозаправки или перезарядки), но в отличие от автомобиля у него может не быть колес (например, у такого транспортного средства, как лодка, очевидно нет колес). Мы говорим, что транспортное средство — это *суперкласс* (superclass) автомобиля, а автомобиль — это *производный класс* (subclass) транспортного средства. У класса транспортного средства может быть несколько производных классов: автомобили, лодки, планеры, мотоциклы, велосипеды и т.д. У производных

классов, в свою очередь, могут быть следующие производные классы. Например, у производного класса лодки могут быть дальнейшие производные классы парусной яхты, гребной шлюпки, каноэ, буксира, моторной лодки и т.д.

В этой главе мы будем использовать пример автомобиля, поскольку это реальный объект, с которым все мы, очевидно связаны.

Создание класса и экземпляра

До ES6 создание классов в JavaScript было суетным и не интуитивно понятным делом. Теперь появился новый удобный синтаксис создания классов.

```
class Car {
    constructor() {
    }
}
```

Это создает новый класс по имени `Car`. Никаких его экземпляров (конкретных автомобилей) еще не создано, но теперь есть возможность сделать это. Чтобы создать конкретный автомобиль, мы используем ключевое слово `new`.

```
const car1 = new Car();
const car2 = new Car();
```

Теперь у нас есть два экземпляра класса `Car`. Прежде чем сделать класс `Car` более сложным, давайте рассмотрим оператор `instanceof`, который может сказать вам, является ли данный объект экземпляром данного класса.

```
car1 instanceof Car   // true
car1 instanceof Array // false
```

Из этого видно, что `car1` — экземпляр класса `Car`, а `Array` — нет.

Давайте сделаем класс `Car` немного интереснее. Придадим ему некие данные (марка, модель) и некие функции (переключение передач).

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this.userGears = ['P', 'N', 'R', 'D'];
        this.userGear = this.userGears[0];
    }
    shift(gear) {
        if(this.userGears.indexOf(gear) < 0)
            throw new Error('Ошибочная передача: ${gear}');
        this.userGear = gear;
    }
}
```

Здесь ключевое слово `this` используется по прямому назначению: для обращения к экземпляру, метод которого был вызван. Вы можете считать его знакоместом:

когда вы пишете свой класс, вероятно, абстрактный, ключевое слово this является знакоместом для *конкретного* экземпляра, который будет известен на момент вызова метода. Этот конструктор позволяет задать марку и модель автомобиля при его создании, а также установить некоторые стандартные значения: допустимые передачи (userGears) и текущую передачу (gear), которую мы инициализируем значением первой допустимой передачи. (Я решил назвать это *пользовательскими* передачами (user gears) потому, что если этот автомобиль оснащен автоматической коробкой передач, то, когда автомобиль будет находиться в движении, фактически использующаяся передача может отличаться от включенной пользователем.) Кроме конструктора (который вызывается неявно при создании нового объекта), мы также создали метод shift, позволяющий переключать передачу. Давайте рассмотрим это в действии.

```
const car1 = new Car("Tesla", "Model S");
const car2 = new Car("Mazda", "3i");
car1.shift('D');
car2.shift('R');
```

В этом примере, когда мы вызываем car1.shift('D'), переменная this связана с car1. Точно так же при вызове car2.shift('R') она связана с car2. Мы можем убедиться, что car1 находится в движении на передаче D (drive), а car2 сдает назад на передаче R (reverse).

```
> car1.userGear // "D"
> car2.userGear // "R"
```

Динамические свойства

То, что метод shift нашего класса Car предотвращает выбор недопустимой передачи по небрежности, может казаться очень умным ходом. Но эта защита ограничена, поскольку нет ничего, что помешало бы установить значение непосредственно: car1.userGear = 'X'. Большинство объектно-ориентированных языков идет на большие затраты, чтобы предоставить механизмы защиты от этого вида неправильного обращения, разрешая вам определять уровень доступа к методам и свойствам. В JavaScript такого механизма нет, за что его нередко и критикуют.

Динамические свойства[1] способны несколько сгладить этот недостаток. Они обладают семантикой свойств с функциональными возможностями методов. Давайте изменим наш класс Car так, чтобы использовать это в своих интересах.

```
class Car {
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this._userGears = ['P', 'N', 'R', 'D'];
        this._userGear = this._userGears[0];
```

[1] Динамические свойства было бы правильнее называть *методами доступа к свойствам* (accessor properties), о которых мы узнаем больше в главе 21.

```
    }

    get userGear() { return this._userGear; }
    set userGear(value) {
        if(this._userGears.indexOf(value) < 0)
            throw new Error('Ошибочная передача: ${value}');
        this._userGear = value;
    }

    shift(gear) { this.userGear = gear; }
}
```

Проницательный читатель заметил, что мы не устранили проблему, поскольку значение `_userGear` все еще можно установить непосредственно: `car1._userGear = 'X'`. В этом примере мы используем "ограничение доступа для бедных" — свойства, имена которых начинаются с символа подчеркивания, мы считаем закрытыми. Эта защита сугубо в соглашении, позволяющем быстро просмотреть код и выявить свойства, к которыми вы не должны обращаться непосредственно.

Если вы действительно должны обеспечить конфиденциальность, то можете использовать экземпляр `WeakMap` (см. главу 10), который защищен областью видимости (если мы не будем использовать `WeakMap`, то наши закрытые свойства никогда не будут выходить из области видимости, даже если экземпляры, к которым они относятся, выйдут). Чтобы сделать основное текущее свойство передачи действительно закрытым, мы можем изменить свой класс `Car` так.

```
const Car = (function() {

    const carProps = new WeakMap();

    class Car {
        constructor(make, model) {
            this.make = make;
            this.model = model;
            this._userGears = ['P', 'N', 'R', 'D'];
            carProps.set(this, { userGear: this._userGears[0] });
        }

        get userGear() { return carProps.get(this).userGear; }
        set userGear(value) {
            if(this._userGears.indexOf(value) < 0)
                throw new Error('Ошибочная передача: ${value}');
            carProps.get(this).userGear = value;
        }

        shift(gear) { this.userGear = gear; }
    }

    return Car;
})();
```

Чтобы поместить наш `WeakMap` в замкнутое выражение, к которому нельзя обратиться извне, мы используем немедленно вызываемое функциональное выражение (см. главу 13). Теперь `WeakMap` может безопасно хранить любые свойства, к которым мы не хотим обращаться за пределами класса.

Существует и другой способ, который подразумевает использование символов для имен свойств; они также предоставляют некоторую защиту от случайного использования, но к символьным свойствам класса также можно обратиться, а значит, даже эту защиту можно обойти.

Классы как функции

До введения в ES6 ключевого слова `class` для создания класса приходилось создавать функцию, которая служила бы конструктором класса. Хотя синтаксис `class` намного более интуитивно понятен и прост, внутренний характер классов в JavaScript не изменился (ключевое слово `class` лишь обеспечивает немного более удобный синтаксис), поэтому важно понимать, что именно представляет собой класс в JavaScript.

В действительности класс — это только функция. В ES5 мы начали бы свой класс `Car` так.

```
function Car(make, model) {
    this.make = make;
    this.model = model;
    this._userGears = ['P', 'N', 'R', 'D'];
    this._userGear = this.userGears[0];
}
```

Мы все еще можем сделать это и в ES6 — результат будет тот же (до методов мы дойдем ниже). Мы можем проверить это, опробовав оба пути.

```
class Es6Car {}       // опустим конструктор для краткости
function Es5Car {}
> typeof Es6Car       // "function"
> typeof Es5Car       // "function"
```

Таким образом, ничего действительно нового в ES6 нет; у нас есть только некий новый удобный синтаксис.

Прототип

Когда говорят о методах, доступных в экземплярах класса, имеют в виду *прототип* (prototype) методов. Например, упоминая метод `shift`, доступный в экземплярах класса `Car`, вы имеете в виду прототип метода и зачастую можете встретить синтаксис `Car.prototype.shift`. (Точно так же функция `forEach` класса `Array` может выглядеть как `Array.prototype.forEach`.) Теперь пришло время фактически узнать, что такое прототип и как JavaScript осуществляет *динамический вызов* (dynamic dispatch), используя *цепь прототипов* (prototype chain).

Использование знака диеза (#) стало популярным соглашением для описания прототипов методов. Например, вы будете часто встречать `Car.prototype.shift`, записанный просто как `Car#shift`.

Каждая функция имеет специальное свойство `prototype`. (Вы можете изменить его для любой функции `f`, введя на консоли **`f.prototype`**.) Для обычных функций прототип не используется, но он критически важен для функций, которые действуют как конструкторы объектов.

В соответствии с соглашением имена конструкторов объектов (иначе — классов) всегда начинаются с заглавной буквы, например `Car`. Это соглашение — не догма, но многие анализаторы предупредят вас, если вы попытаетесь называть функцию с заглавной буквы или конструктор объекта — со строчной.

Свойство функции `prototype` становится важным, когда вы создаете новый экземпляр с использованием ключевого слова `new`: вновь созданный объект имеет доступ к свойству `prototype` его конструктора. Экземпляр объекта хранит его в своем свойстве `__proto__`.

Свойство `__proto__` считается внутренней частью JavaScript, как и любое свойство, заключенное между двойными символами подчеркивания. Используя эти свойства, можно сделать очень, очень много вреда. Иногда их можно использовать очень хитро и правильно, но пока у вас нет полного понимания JavaScript, я настоятельно рекомендую только просматривать (но не изменять) эти свойства.

В прототипе важнее всего механизм *динамического вызова* (термин "dispatch" — это синоним вызова метода). Когда вы пытаетесь получить доступ к свойству или методу объекта, если его не существует, JavaScript *проверяет прототип объекта*, чтобы убедиться, есть ли он там. Поскольку все экземпляры данного класса совместно используют один и тот же прототип, к свойству или методу, имеющемуся в прототипе, есть доступ для всех экземпляров этого класса.

Присвоение значения свойствам данных в прототипе класса обычно не выполняется. Дело в том что тогда значение этого свойства будет доступно для всех экземпляров класса. Однако если значение свойства *устанавливается* в каком-нибудь экземпляре, оно устанавливается именно в этом экземпляре, а не в прототипе, чтобы избежать путаницы и ошибок. Если экземпляры должны иметь начальные значения свойств данных, то лучше устанавливать их в конструкторе.

Обратите внимание, что определение метода или свойства в экземпляре переопределяет версию в прототипе; помните, что JavaScript сначала проверяет экземпляр, а только затем — прототип. Давайте рассмотрим все это на примере.

```
// определенный ранее класс Car с методом shift
const car1 = new Car();
const car2 = new Car();
car1.shift === Car.prototype.shift; // true
car1.shift('D');
car1.shift('d');                    // ошибка
car1.userGear;                      // 'D'
car1.shift === car2.shift           // true
car1.shift = function(gear) { this.userGear = gear.toUpperCase(); }
car1.shift === Car.prototype.shift; // false
car1.shift === car2.shift;          // false
car1.shift('d');
car1.userGear;                      // 'D'
```

В этом примере ясно показано, как JavaScript осуществляет динамический вызов. Первоначально у объекта `car1` нет метода `shift`, но при вызове `car1.shift('D')` JavaScript просматривает прототип для `car1` и находит метод с таким именем. Когда мы заменяем метод `shift` собственной версией, то у объекта `car1` и у его прототипа появляется метод с этим именем. Однако при вызове `car1.shift('d')`, будет вызван метод объекта `car1`, а не его прототипа.

Обычно в знании механики цепи прототипов и динамического вызова у вас не будет особой нужды, но все же может встретиться проблема, которая потребует их глубокого понимания. Поэтому, прежде чем продолжать, имеет смысл узнать детали.

Статические методы

Методы, которые мы рассматривали до сих пор, являлись *методами экземпляра* (instance method). Они предназначены для работы с конкретным экземпляром. Есть также *статические методы* (static method) (или *методы класса* (class method)), которые не относятся ни к какому конкретному экземпляру. В статическом методе переменная `this` привязана к самому классу, но в этом случае вместо нее рекомендуется использовать имя класса.

Статические методы используются для выполнения обобщенных задач, которые связаны с классом, а не с любым конкретным экземпляром. Давайте рассмотрим пример использования VIN автомобиля (идентификатор транспортного средства). Нет смысла позволять индивидуальному автомобилю создавать собственный VIN: что помешало бы автомобилю использовать такой же VIN, как и у другого автомобиля? Однако присвоение VIN является абстрактной концепцией, которая связана с идеей автомобиля вообще; следовательно, это кандидат в статические методы. Кроме того, статические методы зачастую используются для работы с несколькими транспортными средствами (объектами). Например, нам может понадобиться метод `areSimilar`, который возвращает `true`, если у двух автомобилей те же марка и модель, а метод `areSame`, возвращающий `true`, если у двух автомобилей один и тот же VIN. Давайте рассмотрим эти статические методы, реализованные для класса `Car`.

```
class Car {
    static getNextVin() {
        return Car.nextVin++; // мы могли бы также использовать
                              // this.nextVin++, но обращение к Car
                              // подчеркивает, что это статический метод
    }
    constructor(make, model) {
        this.make = make;
        this.model = model;
        this.vin = Car.getNextVin();
    }
    static areSimilar(car1, car2) {
        return car1.make===car2.make && car1.model===car2.model;
    }
    static areSame(car1, car2) {
        return car1.vin===car2.vin;
    }
}
Car.nextVin = 0;

const car1 = new Car("Tesla", "S");
const car2 = new Car("Mazda", "3");
const car3 = new Car("Mazda", "3");

car1.vin;  // 0
car2.vin;  // 1
car3.vin   // 2

Car.areSimilar(car1, car2);  // false
Car.areSimilar(car2, car3);  // true
Car.areSame(car2, car3);     // false
Car.areSame(car2, car2);     // true
```

Наследование

Рассматривая прототипы, мы уже встречали некий вид наследования: при создании экземпляра класса он наследовал все функции, находящиеся в прототипе класса. Но на этом дело не заканчивается: если метод не найден в прототипе объекта, проверяется прототип *прототипа.* Так получается *цепь прототипов.* JavaScript будет идти по цепи прототипов, пока не найдет тот прототип, который удовлетворяет запросу. Если такой прототип не будет найден, то все закончится ошибкой.

Это пригодится при создании иерархии классов. Мы уже упоминали, что автомобиль — это общий тип транспортного средства. Цепь прототипов позволяет располагать функции там, где им самое место. Например, у автомобиля мог бы быть метод `deployAirbags`. Мы могли бы сделать его методом обобщенного транспортного средства, но вы когда-либо видели лодку с подушками безопасности? С другой стороны, почти все транспортные средства могут перевозить пассажиров; таким образом, у транспортного средства мог бы быть метод `addPassenger` (который мог бы

сообщать об ошибке, если количество пассажиров превышено). Давайте посмотрим, как этот сценарий реализуется в коде JavaScript.

```
class Vehicle {
    constructor() {
        this.passengers = [];
        console.log("Транспортное средство создано");
    }
    addPassenger(p) {
        this.passengers.push(p);
    }
}

class Car extends Vehicle {
    constructor() {
        super();
        console.log("Автомобиль создан");
    }
    deployAirbags() {
        console.log("БАБАХ!!!");
    }
}
```

Первое нововведение, которое мы замечаем, — это ключевое слово `extends`; этот синтаксис указывает, что класс `Car` происходит от класса `Vehicle`. Второй новостью является вызов `super()`. Это специальная функция JavaScript, которая вызывает конструктор суперкласса. Для производных классов это обязательно; если вы опустите его, то получите ошибку.

Давайте рассмотрим этот пример в действии.

```
const v = new Vehicle();
v.addPassenger("Frank");
v.addPassenger("Judy");
v.passengers;                // ["Frank", "Judy"]
const c = new Car();
c.addPassenger("Alice");
c.addPassenger("Cameron");
c.passengers;                // ["Alice", "Cameron"]
v.deployAirbags();           // ошибка
c.deployAirbags();           // "БАБАХ!!!"
```

Обратите внимание, что мы можем вызвать метод `deployAirbags` с `c`, но не с `v`. Другими словами, наследование работает только в одном направлении. Экземпляры класса `Car` могут обращаться ко всем методам класса `Vehicle`, но не наоборот.

Полиморфизм

Термин *полиморфизм* (polymorphism) из лексикона объектно-ориентированных языков описывает ситуацию, когда экземпляр рассматривается как член не только

его собственного класса, но и любых суперклассов. На многих объектно-ориентированных языках полиморфизм — это нечто особенное, приносящее большую пользу ООП. Язык JavaScript не является типизированным, т.е. любой объект может быть использован в любом месте (хотя правильный результат не гарантирован). Таким образом, в некотором смысле у JavaScript есть абсолютный полиморфизм.

Код JavaScript, который вы пишете, довольно часто использует некую форму *утиной типизации* (duck typing). Эта методика исходит из выражения "Если это выглядит, как утка, плавает, как утка и крякает, как утка, то это, возможно, и есть утка". В нашем примере с классом `Car`, если у вас есть объект, обладающий методом `deployAirbags`, то вы могли бы резонно заключить, что это экземпляр класса `Car`. Это может быть правда, а может и нет, но попытка довольно хорошая.

В JavaScript предусмотрен оператор `instanceof`, который укажет вам, является ли объект экземпляром данного класса. Как ни удивительно, но до тех пор, пока вы не оперируете напрямую свойствами `prototype` и `__proto__`, этот оператор будет возвращать правильный результат.

```
class Motorcycle extends Vehicle {}
const c = new Car();
const m = new Motorcyle();
c instanceof Car;            // true
c instanceof Vehicle;        // true
m instanceof Car;            // false
m instanceof Motorcycle;     // true
m instanceof Vehicle;        // true
```

Все объекты в JavaScript являются экземплярами корневого класса `Object`. Таким образом, для любого объекта `o` выражение `o instanceof Object` будет истинным (если только вы явно не установите значение его свойства `__proto__`, чего следует избегать). С практической точки зрения в этом есть небольшой смысл, поскольку такая возможность позволяет создать ряд важных методов для всех объектов иерархии. В качестве примера можно привести метод `toString`, который будет рассмотрен ниже в этой главе.

Перебор свойств объектов (снова)

Мы уже видели, как можно перебрать свойства объекта в цикле `for...in`. Теперь, когда мы понимаем механизм наследования прототипов, мы можем полностью оценить использование метода `hasOwnProperty` при переборе свойств объекта. Для объекта `obj` и свойства `x` вызов `obj.hasOwnProperty(x)` возвратит `true`, если у `obj` будет свойство `x`, и `false`, если свойство `x` не определено или определено в цепи прототипов.

Если вы будете использовать классы ES6 так, как они задуманы, то свойства данных всегда будут определяться в экземплярах, а не в цепи прототипов. Однако,

поскольку нет ничего, что предотвратило бы добавление свойств непосредственно в прототип, всегда лучше использовать `hasOwnProperty`, чтобы удостовериться в этом. Рассмотрим пример.

```
class Super {
    constructor() {
        this.name = 'Super';
        this.isSuper = true;
    }
}

// это допустимо, но не желательно...
Super.prototype.sneaky = 'Не рекомендуется!';

class Sub extends Super {
    constructor() {
        super();
        this.name = 'Sub';
        this.isSub = true;
    }
}

const obj = new Sub();

for(let p in obj) {
    console.log('${p}: ${obj[p]}' +
         (obj.hasOwnProperty(p) ? '' : ' (унаследовано)'));
}
```

Если вы запустите эту программу, то увидите

```
name: Sub
isSuper: true
isSub: true
sneaky: Не рекомендуется! (унаследовано)
```

Свойства `name`, `isSuper` и `isSub` определяются в экземпляре, а не в цепи прототипов (обратите внимание, что свойства, объявленные в конструкторе суперкласса, присутствуют также в экземпляре производного класса). Свойство `sneaky`, напротив, было вручную добавлено в прототип суперкласса.

Вы можете избежать этой проблемы в целом, используя метод `Object.keys`, который включает только свойства, определенные в прототипе.

Строковое представление

Каждый объект в конечном счете происходит от класса `Object`. Таким образом, все методы, доступные в классе `Object`, стандартно доступны для всех объектов. Одним из этих методов является `toString`, возвращающий стандартное строковое

представление объекта. Стандартное поведение метода `toString` подразумевает возвращение строки `"[object Object]"`, что не особенно полезно.

Наличие метода `toString`, выводящего нечто описательное об объекте, может очень пригодиться при отладке, позволяя сразу получить важную информацию об объекте. Например, мы могли бы изменить свой класс `Car` так, чтобы его метод `toString` возвращал марку, модель и VIN.

```
class Car {
    toString() {
        return '${this.make} ${this.model}: ${this.vin}';
    }
    //...
```

Теперь вызов метода `toString` для экземпляра `Car` дает немного больше информации об объекте.

Множественное наследование, примеси и интерфейсы

Некоторые объектно-ориентированные языки поддерживают *множественное наследование* (multiple inheritance), когда у одного класса может быть два прямых суперкласса (в отличие от одного суперкласса, у которого, в свою очередь, есть один суперкласс). Множественное наследование создает риск *коллизий* (collision) или конфликтов. Таким образом, если нечто унаследовано от двух родителей и у обоих родителей есть метод `greet`, то от кого именно он будет унаследован производным классом? Во многих языках предпочитается одиночное наследование, при котором этой проблемы нет.

Но когда мы решаем реальные задачи, множественное наследование зачастую имеет смысл. Например, автомобили могли бы происходить как от транспортных средств, так и от "подлежащих страхованию" (вы можете застраховать и автомобиль, и дом, но дом, безусловно, — не транспортное средство). В языках, где не поддерживается множественное наследование, зачастую вводится концепция *интерфейса* (interface), чтобы обойти эту проблему. Класс (`Car`) может происходить только от одного родителя (`Vehicle`), но у него может быть несколько интерфейсов (`Insurable`, `Container` и т.д.).

JavaScript — интересный гибрид. Технически это язык одиночного наследования, поскольку поиск по цепи прототипов не распространяется на несколько родителей, но он предоставляет пути, которые иногда превосходят и множественное наследование, и интерфейсы (а иногда — нет).

Основной механизм решения проблемы множественного наследования — это концепция *примеси* (mixin). Примесь позволяет "подмешивать" функциональные возможности по мере необходимости. Поскольку JavaScript позволяет чрезвычайно много и без контроля типов, вы можете подмешать почти любые функции к любому объекту в любое время.

Давайте создадим примесь "страхуемый", которую мы могли бы применить к автомобилям. Мы не будем усложнять пример, но в дополнение к примеси страхования необходимо создать класс `InsurancePolicy`. Примесь страхования нуждается в методах `addInsurancePolicy`, `getInsurancePolicy` и (для удобства) `isInsured`. Давайте рассмотрим, как это могло бы работать.

```
class InsurancePolicy() {}
function makeInsurable(o) {
    o.addInsurancePolicy = function(p) { this.insurancePolicy = p; }
    o.getInsurancePolicy = function() { return this.insurancePolicy; }
    o.isInsured = function() { return !!this.insurancePolicy; }
}
```

Теперь мы можем сделать любой объект подлежащим страхованию. Так как мы собираемся сделать подлежащим страхованию класс `Car`? Ваша первая мысль могла бы быть такой.

```
makeInsurable(Car);
```

Но вы были бы неприятно удивлены.

```
const carl = new Car();
carl.addInsurancePolicy(new InsurancePolicy()); // ошибка
```

Если вы подумали "Конечно, ведь `addInsurancePolicy` не находится в цепи прототипов", то будете совершенно правы. Делать класс `Car` подлежащим страхованию вообще плохая идея. Кроме того, это вообще не имеет смысла: абстрактная концепция автомобиля не подлежит страхованию, но конкретный автомобиль — подлежит. Таким образом, наше следующее решение могло бы быть таким.

```
const carl = new Car();
makeInsurable(carl);
carl.addInsurancePolicy(new InsurancePolicy()); // работает
```

Это работает, но теперь нужно не забыть вызывать функцию `makeInsurable` для каждого создаваемого нами автомобиля. Мы могли бы добавить этот вызов в конструктор `Car`, но тогда мы продублируем эту функцию для каждого созданного автомобиля. К счастью, решение простое.

```
makeInsurable(Car.prototype);
const carl = new Car();
carl.addInsurancePolicy(new InsurancePolicy()); // работает
```

Теперь это выглядит так, как будто наши методы всегда были частью класса `Car`. И с точки зрения JavaScript *так и есть*. С точки зрения разработчика мы облегчили поддержку этих двух важных классов. Группа разработчиков автомобилей создает и обслуживает класс `Car`, а группа страхования занимается классом `InsurancePolicy` и примесью `makeInsurable`. В результате место для пересечения двух групп все таки имеется, но это куда лучше, чем когда все работают над одним гигантским классом `Car`.

Примеси не устраняют проблему коллизий: если бы по каким-то причинам страховая группа должна была бы создать в своей примеси метод `shift`, то это нарушило бы класс `Car`. Кроме того, мы не можем использовать `instanceof` для выявления объектов, которые допускают страхование: в лучшем случае мы можем рассчитывать на утиную типизацию (если у объекта есть метод `addInsurancePolicy`, он, вероятно, подлежит страхованию).

Мы можем сгладить некоторые из этих проблем, используя символы. Скажем, страховая группа постоянно добавляет очень обобщенные методы, которые конфликтуют с методами класса `Car`. Вы могли бы попросить их использовать для всех своих ключей символы. Теперь их примесь будет выглядеть следующим образом.

```
class InsurancePolicy() {}
const ADD_POLICY = Symbol();
const GET_POLICY = Symbol();
const IS_INSURED = Symbol();
const _POLICY = Symbol();
function makeInsurable(o) {
    o[ADD_POLICY] = function(p) { this[_POLICY] = p; }
    o[GET_POLICY] = function() { return this[_POLICY]; }
    o[IS_INSURED] = function() { return !!this[_POLICY]; }
}
```

Поскольку символы уникальны, это гарантирует, что примесь никогда не будет конфликтовать с существующими функциями класса `Car`. В использовании это может быть немного неуклюже, но намного безопаснее. Компромиссный подход, возможно, подразумевал бы использование обычных строк для методов, а символов (таких, как `_POLICY`) для свойств данных.

Заключение

Объектно-ориентированное программирование — это чрезвычайно популярная парадигма и на то есть серьезные причины. Оно обеспечивает организацию и инкапсуляцию кода для решения многих реальных задач, что облегчает поддержку, отладку и исправление ошибок. Реализацию ООП в JavaScript серьезно критикуют — некоторые доходят до утверждения, что JavaScript даже не соответствует определению объектно-ориентированного языка (обычно из-за нехватки контроля доступа к данным). Этот аргумент заслуживает внимания, но как только вы привыкнете к тому, как в JavaScript реализовано ООП, вы найдете этот язык весьма гибким и мощным. Он позволяет делать такие вещи, на которые другие объектно-ориентированные языки не способны.

Отображения и наборы

В ES6 введены две популярные структуры данных: *отображения* (map) и *наборы* (set). Отображения подобны объектам, они способны сопоставлять ключи со значениями, а наборы подобны массивам за исключением того, что дубликаты не допускаются.

Отображения

До появления ES6, когда требовалось сопоставить ключам значения, использовались объекты, поскольку объекты позволяют сопоставить строковые ключи со значениями объектов любых типов. Однако при использовании объектов для этой цели возникает много проблем.

- Прототипы, лежащие в основе объектов, способны создать сопоставления, о которых вы и не предполагали.
- Нет никакого простого способа узнать количество сопоставлений, находящихся в объекте.
- Ключи должны быть строками или символами, сопоставить со значениями объекты невозможно.
- Объекты не гарантируют порядка своих свойств.

Объект `Map` ликвидирует эти недочеты и является превосходным выбором для сопоставления ключей со значениями (даже если ключи — строки). Предположим, например, что у вас есть объекты пользователей, которые необходимо сопоставить с ролями.

```
const u1 = { name: 'Cynthia' };
const u2 = { name: 'Jackson' };
const u3 = { name: 'Olive' };
const u4 = { name: 'James' };
```

Сначала создадим отображение.

```
const userRoles = new Map();
```

Затем используем отображение для назначения пользователям ролей с использованием ее метода `set()`.

```
userRoles.set(u1, 'User');
userRoles.set(u2, 'User');
userRoles.set(u3, 'Admin');
// бедный Джеймс... мы не назначили ему роль
```

Метод `set()` допускает также цепочки, что позволяет сэкономить на вводе.

```
userRoles
    .set(u1, 'User')
    .set(u2, 'User')
    .set(u3, 'Admin');
```

Вы можете также передать в конструктор массив массивов.

```
const userRoles = new Map([
    [u1, 'User'],
    [u2, 'User'],
    [u3, 'Admin'],
]);
```

Теперь, если необходимо выяснить роль пользователя `u2`, можно использовать метод `get()`.

```
userRoles.get(u2);    // "User"
```

Вызов метода `get` для ключа, отсутствующего в отображении, возвратит значение `undefined`. Кроме того, вы можете использовать метод `has()` для определения наличия в отображении заданного ключа.

```
userRoles.has(u1);    // true
userRoles.get(u1);    // "User"
userRoles.has(u4);    // false
userRoles.get(u4);    // undefined
```

Вызов метода `set()` для ключа, уже присутствующего в отображении, приведет к замене его значения.

```
userRoles.get(u1);          // 'User'
userRoles.set(u1, 'Admin');
userRoles.get(u1);         // 'Admin'
```

Свойство `size` возвращает количество элементов в отображении.

```
userRoles.size;            // 3
```

Метод `keys()` позволяет получить ключи в отображении, метод `values()` — возвратить значения, а метод `entries()` — получить элементы в виде массивов, в которых первый элемент — ключ, а второй — значение. Все эти методы возвращают итерируемый объект, который может быть перебран в цикле `for...of`.

```
for(let u of userRoles.keys())
    console.log(u.name);

for(let r of userRoles.values())
    console.log(r);

for(let ur of userRoles.entries())
    console.log('${ur[0].name}: ${ur[1]}');

// обратите внимание: чтобы сделать этот перебор еще более
// естественным, мы можем использовать деструктуризацию:
for(let [u, r] of userRoles.entries())
    console.log('${u.name}: ${r}');

// метод entries() - это стандартный итератор для отображений, так
// вы можете сократить предыдущий пример:
for(let [u, r] of userRoles)
    console.log('${u.name}: ${r}');
```

Если вместо итерируемого объекта необходим массив, вы можете использовать оператор расширения.

```
[...userRoles.values()];    // [ "User", "User", "Admin" ]
```

Чтобы удалить одиночный элемент из отображения, используйте метод `delete()`.

```
userRoles.delete(u2);
userRoles.size;             // 2
```

Наконец, если вы хотите удалить все элементы из отображения, то можете сделать это, используя метод `clear()`.

```
userRoles.clear();
userRoles.size;             // 0
```

Слабые отображения

Объект `WeakMap` идентичен объекту `Map`, кроме следующего.

- Его ключи должны быть объектами.
- Ключи в `WeakMap` допускают сборку мусора.
- Объект `WeakMap` не может быть перебран или очищен.

Обычно JavaScript хранит объект в памяти, пока где-нибудь есть ссылка на него. Например, если у вас будет объект, который является ключом в `Map`, то JavaScript будет хранить этот объект в памяти, пока объект `Map` существует. С `WeakMap` все не так. Из-за этого объект `WeakMap` не может быть перебран (есть слишком большая

опасность, что при переборе произойдет доступ к объекту, который уже был уничтожен в процессе сборки мусора).

Благодаря этим свойствам объект `WeakMap` применяется для хранения закрытых ключей в экземплярах объекта.

```
const SecretHolder = (function() {
    const secrets = new WeakMap();
    return class {
        setSecret(secret) {
            secrets.set(this, secret);
        }
        getSecret() {
            return secrets.get(this);
        }
    }
})();
```

Здесь мы поместили свой объект `WeakMap` в немедленно вызываемое функциональное выражение (IIFE) наравне с классом, который его использует. Вне IIFE мы получаем класс `SecretHolder`, экземпляры которого способны хранить секреты. Мы можем установить секрет, только используя метод `setSecret`, а получить к нему доступ — только через метод `getSecret`.

```
const a = new SecretHolder();
const b = new SecretHolder();

a.setSecret('secret A');
b.setSecret('secret B');

a.getSecret();    // "secret A"
b.getSecret();    // "secret B"
```

Мы могли бы использовать обычный объект `Map`, но сообщенные его экземплярам `SecretHolder` секреты никогда не будут уничтожены в процессе сборки мусора!

Наборы

Набор (set) — это коллекция данных, в которой дубликаты недопустимы. Используя наш предыдущий пример, мы можем назначить пользователя на несколько ролей. Например, у всех пользователей могла бы быть роль `"User"`, а у администраторов — и `"User"`, и `"Admin"`. Однако для пользователя нет никакого логического смысла иметь одну и ту же роль многократно. Набор — идеальная структура данных для этого случая.

Сначала создайте экземпляр объекта `Set`.

```
const roles = new Set();
```

Если мы теперь хотим добавить роль пользователя, можем воспользоваться методом `add()`.

```
roles.add("User");     // Набор [ "User" ]
```

Чтобы сделать этого пользователя администратором, вызовите метод `add()` снова.

```
roles.add("Admin");    // Набор [ "User", "Admin" ]
```

Как и у `Map`, у объекта `Set` есть свойство `size`.

```
roles.size;            // 2
```

Достоинство наборов в том, что мы не должны выяснять, находится ли уже нечто в наборе, прежде чем его добавим. При попытке добавить в набор нечто, что уже там находится, ничего не происходит.

```
roles.add("User");     // Набор [ "User", "Admin" ]
roles.size;            // 2
```

Чтобы удалить роль, мы просто вызываем метод `delete()`, который возвращает `true`, если роль была в наборе, и `false` — в противном случае.

```
roles.delete("Admin");     // true
roles;                     // Набор [ "User" ]
roles.delete("Admin");     // false
```

Слабые наборы

Слабые наборы могут содержать только объекты, и эти объекты удаляются в процессе сборки мусора. Как и в `WeakMap`, значения в `WeakSet` не могут быть перебраны, что делает слабые наборы очень редко применяемыми. Фактически единственный подходящий случай использования для слабых наборов — это когда необходимо определять, есть ли данный объект в наборе.

Например, у Санта Клауса мог бы быть `WeakSet` по имени `naughty` (непослушные), чтобы он мог решить, кому достанется уголь.

```
const naughty = new WeakSet();

const children = [
    { name: "Suzy" },
    { name: "Derek" },
];

naughty.add(children[1]);

for(let child of children) {
    if(naughty.has(child))
```

```
        console.log('Уголь для ${child.name}!');
    else
        console.log('Подарки для ${child.name}!');
}
```

Расставаясь с объектной привычкой[1]

Если вы — опытный программист JavaScript, который является новичком в ES6, возможно, вы уже привыкли использовать объекты для сопоставления значений. И без сомнения, вы изучили все нюансы применения объектов в виде отображений, позволяющие обойти подводные камни. Но теперь у вас есть реальные отображения, и вы должны использовать их! Аналогично вы, вероятно, привыкли использовать объекты с логическими значениями в качестве наборов; вам также больше не нужно делать это. Когда вы создаете объект, остановитесь и спросите себя: "Я использую этот объект, только чтобы получить отображение?" Если ответ — "Да", то рассмотрите возможность использования вместо него объекта `Map`.

[1] Перефразировка "Breaking the Habit" рок-группы Linkin Park. — *Примеч. ред.*

Исключения и обработка ошибок

Все мы хотели бы жить в мире без ошибок, но у нас такой роскоши нет. Большинство даже тривиальных приложений подвержено ошибкам, являющимся результатом обстоятельств, которые вы не предвидели. Первый шаг к созданию надежного, высококачественного программного обеспечения — это признать, что в нем будут ошибки. Второй шаг — предвидение этих ошибок и их обработка разумным способом.

Обработка исключений (exception handling) — это механизм, который позволяет справляться с ошибками контролируемым способом. Обработка исключений, в отличие от *обработки ошибок* (error handling), предназначена, чтобы справляться с исключительными обстоятельствами, т.е. не с теми ошибками, которые вы ожидаете, а с непредвиденными.

Грань между ожидаемыми и непредвиденными ошибками (исключениями) весьма расплывчата и очень ситуативна. От приложения, которое предназначено для использования широкой неподготовленной публикой, можно ожидать намного более непредсказуемого поведения, чем от приложения, предназначенного для использования квалифицированными пользователями.

Примером ожидаемой ошибки является ввод в форме неправильного адреса электронной почты: люди *все время* делают опечатки. Непредвиденной ошибкой могло бы быть исчерпание дискового пространства или невозможность доступа к обычно всегда работающей службе.

Объект `Error`

В JavaScript есть встроенный объект `Error`, который удобен для обработки ошибок любого вида (исключений и ожидаемых). Создавая экземпляр объекта `Error`, вы можете присвоить ему сообщение об ошибке.

```
const err = new Error('Ошибочный email');
```

Создание экземпляра `Error` само по себе ничего не делает. Оно лишь предоставляет вам средство для сообщения об ошибке. Вообразите функцию, которая проверяет адреса электронной почты. Если функция сработала успешно, она возвращает

адрес электронной почты как строку. Если это не так, она возвращает экземпляр объекта `Error`. Для простоты будем считать нечто, содержащее символ `@`, допустимым адресом электронной почты (см. главу 17).

```
function validateEmail(email) {
    return email.match(/@/) ?
        email :
        new Error('Ошибочный email: ${email}');
}
```

Чтобы определить, был ли возвращен экземпляр объекта `Error`, мы можем использовать оператор `typeof`. Предоставленное нами сообщение об ошибке будет присвоено свойству `message`.

```
const email = "jane@doe.com";

const validatedEmail = validateEmail(email);
if(validatedEmail instanceof Error) {
    console.error('Ошибка: ${validatedEmail.message}');
} else {
    console.log('Корректный email: ${validatedEmail}');
}
```

Хотя это вполне допустимый и полезный способ использования экземпляра объекта `Error`, он чаще используется в процессе обработки исключений, который мы рассмотрим далее.

Обработка исключений с использованием блоков `try` и `catch`

Для обработки исключений используется конструкция операторов `try...catch`. Идея в том, что осуществляется "попытка" (try) что-то сделать, и если при этом произойдет какое-либо исключение, оно будет "перехвачено" (catch). Функция `validateEmail` в нашем предыдущем примере *обрабатывает* ожидаемую ошибку, когда некто пропускает символ `@` в адресе электронной почты, но есть также возможность возникновения и непредвиденной ошибки: непутевый программист может присвоить переменной `email` нечто отличное от строки. Как следует из предыдущего примера, присвоение переменной `email` значения `null`, числа или объекта (чего угодно, кроме строки) приводит к ошибке. В результате программа сразу же прекращает свое выполнение, что весьма недружественно по отношению к пользователю. Чтобы обезопасить себя от непредвиденной ошибки мы можем поместить свой код в блок оператора `try...catch`.

```
const email = null;       // упс

try {
```

```
    const validatedEmail = validateEmail(email);
    if(validatedEmail instanceof Error) {
        console.error('Ошибка: ${validatedEmail.message}');
    } else {
        console.log('Корректный email: ${validatedEmail}');
    }
} catch(err) {
    console.error('Ошибка: ${err.message}');
}
```

Поскольку мы перехватываем ошибку, наша программа не будет аварийно завершать работу. В данном случае в обработчике ошибок мы просто выводим соответствующее сообщение и продолжаем работу. Однако что делать, если для продолжения работы программы требуется правильный адрес электронной почты? Очевидно, что в таком случае нужно обработать ошибку более изящно и красиво завершить работу программы.

Обратите внимание, что поток выполнения покидает блок `catch`, как только происходит ошибка; т.е. оператор `if`, который следует за вызовом `validateEmail()`, не будет выполнен. В блоке `try` у вас может быть столько операторов, сколько нужно; первый из них, который закончится ошибкой, передаст управление блоку `catch`. Если никаких ошибок нет, блок `catch` не выполняется, и программа продолжает работу.

Генерирование ошибки

В нашем предыдущем примере мы использовали оператор `try...catch` для обработки ошибок, которые возникали в самом движке JavaScript (когда мы пытались вызвать метод `match` для чего-то, что не является строкой). Вы можете также сгенерировать ошибку самостоятельно, чтобы задействовать механизм обработки исключений.

В отличие от других языков с обработкой исключений, в JavaScript при генерации ошибки вы можете использовать любое значение: число, строку или любой другой тип. Однако обычно оператору `throw` передают экземпляр объекта `Error`. Большинство блоков `catch` ожидает экземпляра объекта `Error`. Имейте в виду, что вы не всегда можете контролировать, где будет обработана сгенерированная вами ошибка (функции, которые вы пишете, могут быть использованы другими программистами, вполне резонно ожидающими, что в процессе генерации ошибки оператору `throw` передается экземпляр объекта `Error`).

Например, создавая приложение по оплате счетов для банка, вы могли бы генерировать исключения, если остаток на счете не покрывает платеж (это действительно исключительный случай, поскольку проверка на такую ситуацию должна осуществляться прежде, чем начнется оплата по счету).

```
function billPay(amount, payee, account) {
    if(amount > account.balance)
        throw new Error("Мало денег.");
    account.transfer(payee, amount);
}
```

При выполнении оператора `throw` текущая функция немедленно прекращает свою работу. Поэтому в нашем примере вызова метода `account.transfer` не будет, что нам и требовалось.

Обработка исключений и стек вызовов

Типичная программа вызывает функции, а эти функции, в свою очередь, вызывают другие функции, а эти функции — следующие функции и т.д. Интерпретатор JavaScript должен отслеживать их все. Если функция `a` вызывает функцию `b`, а функция `b` вызывает функцию `c`, то, когда функция `c` завершает работу, управление возвращается функции `b`, а когда завершается функция `b`, управление возвращается функции `a`. Поэтому, когда выполняется функция `c`, функции `a` и `b` "ожидают". Эти вложенные функции, которые еще не завершили работу, формируют *стек вызовов* (call stack).

Если в функции `c` происходит ошибка, то что будет с функциями `a` и `b`? Очевидно, что в функции `b` также возникнет ошибка, поскольку в ней может использоваться значение, возвращаемое функцией `c`. Это в свою очередь вызовет ошибку в функции `a`, поскольку в ней также может использоваться значение, возвращаемое функцией `b`. По существу, ошибка будет распространяться по стеку вызовов вверх, пока не будет перехвачена и обработана.

Ошибки могут быть перехвачены и обработаны на любом уровне в стеке вызовов. Если они так и не будут перехвачены, интерпретатор JavaScript просто остановит программу. Это явление называется *необработанным исключением* (unhandled exception) или *не перехваченным исключением* (uncaught exception), оно всегда приводит к аварийному завершению программы. С учетом количества мест, где может произойти ошибка, перехват всех возможных ошибок, способных привести к аварийному завершению программы, становится трудоемким и громоздким.

Когда ошибка *перехватывается*, стек вызовов предоставляет полезную информацию для диагностики проблемы. Например, если функция `a` вызывает функцию `b`, которая вызывает функцию `c` и ошибка происходит в функции `c`, то стек вызовов говорит нам не только о том, что ошибка произошла в функции `c`, но и что она произошла, когда эта функция была вызвана функцией `b`, когда она была вызвана функцией `a`. Это полезная информация, если функция `c` вызывается из многих разных мест в вашей программе.

В большинстве реализаций JavaScript экземпляры объекта `Error` содержат свойство `stack`, которое является строковым представлением стека (это нестандартное средство JavaScript, но оно доступно в большинстве систем). Вооружившись этими знаниями, мы можем написать пример, который демонстрирует обработку исключений.

```
function a() {
    console.log('a: вызываем b');
    b();
    console.log('a: готово');
}
function b() {
    console.log('b: вызываем c');
    c();
    console.log('b: готово');
}
function c() {
    console.log('c: генерируем ошибку');
    throw new Error('c ошибка');
    console.log('c: готово');
}
function d() {
    console.log('d: вызываем c');
    c();
    console.log('d: готово');
}

try {
    a();
} catch(err) {
    console.log(err.stack);
}

try {
    d();
} catch(err) {
    console.log(err.stack);
}
```

Запуск этого примера в Firefox приводит к следующему выводу на консоль.

```
a: вызываем b
b: вызываем c
c: генерируем ошибку
c@debugger eval code:13:1
b@debugger eval code:8:4
a@debugger eval code:3:4
@debugger eval code:23:4
```

```
d: вызываем c
c: генерируем ошибку
c@debugger eval code:13:1
d@debugger eval code:18:4
@debugger eval code:29:4
```

Строки со знаком `@` означают трассировку стека, которая начинается с "самой глубокой" функции (`c`) и завершается без функции вообще (сам браузер). Как можно заметить, имеется две разных трассировки стека. В первой мы видим, что функция `c` была вызвана из `b`, а та, в свою очередь, была вызвана из `a`. Во второй видно, что функция `c` была вызвана непосредственно из `d`.

Конструкция `try...catch...finally`

Иногда в коде блока `try` задействуется некий ресурс, такой как подключение к серверу HTTP или открытие файла. Вне зависимости от ошибки мы должны освободить этот ресурс, чтобы он не был постоянно связан с нашей программой. Поскольку блок `try` может содержать любое количество операторов, в каждом из которых может возникнуть ошибка, поэтому блок `try` не самое безопасное место для освобождения ресурса (поскольку ошибка может произойти прежде, чем представится шанс сделать это). Также небезопасно освобождать ресурс в блоке `catch`, поскольку он не выполняется, если не будет ошибки. Это именно та ситуация, которая требует блока `finally`, выполняемого вне зависимости от наличия ошибки.

Поскольку мы еще не рассматривали работу с файлами или подключениями к серверу HTTP, для демонстрации блока `finally` мы будем просто использовать пример с операторами `console.log`.

```
try {
    console.log("Эта строка выполнена...");
    throw new Error("Упс!");
    console.log("Эта строка не выполняется...");
} catch(err) {
    console.log("Была ошибка...");
} finally {
    console.log("...всегда выполняется");
    console.log("Здесь выполняется очистка");
}
```

Опробуйте этот пример с оператором `throw` и без него; вы увидите, что блок `finally` выполняется в любом случае.

Позвольте исключениям быть исключениями

Теперь, когда вы знаете, что такое обработка исключений и как ее осуществлять, наверняка вы захотите использовать ее для обработки всех ошибок — как ожидаемых, так и нет. В конце концов, генерирование ошибки, чрезвычайно простой и удобный способ "выхода", когда вы попадаете в ситуацию, с которой не можете справиться. Но обработка исключений имеет свою цену. Кроме риска, что исключение так и не будет перехвачено (это приведет к аварийному завершению программы), применение исключений создает дополнительную вычислительную нагрузку. Поскольку исключения должны "прокрутить" стек, пока не встретится блок `catch`, интерпретатор JavaScript вынужден выполнять некоторые дополнительные служебные действия. При постоянном росте скоростей компьютеров это вызывает все меньше и меньше беспокойства, но генерирование исключений в часто используемых ветках программы может снизить ее производительность.

Помните, что каждый раз, генерируя исключение, вы должны обработать его (если не хотите столкнуться с аварийным завершением программы). Вы не можете получить нечто из ничего. Исключения лучше использовать лишь как последнюю линию обороны, для обработки исключительных ситуаций, которые вы не можете предвидеть, а для исправления ожидаемых ошибок используйте операторы управления потоком.

ГЛАВА 12

Итераторы и генераторы

В спецификацию ES6 введено две очень важные новые концепции: *итераторы* (iterator) и *генераторы* (generator). Генераторы связаны с итераторами, поэтому давайте начнем с итераторов.

Итератор напоминает закладку: он помогает следить, где вы находитесь. Массив — пример *итерируемого* (iterable) объекта: он содержит множество элементов (по аналогии со страницами в книге) и может предоставить итератор (который похож на закладку). Давайте сделаем эту аналогию конкретней: предположим, что у вас есть массив `book` (книга), каждый элемент которого — строка, представляющая страницу. Формату такой книги лучше всего соответствует стишок "Twinkle, Twinkle, Little Bat" (Крокодильчики мои, цветики речные![1]) из *Алисы в стране чудес* Льюиса Кэрролла (представьте, что у вас детская книжка, на каждой странице которой расположена всего одна строчка).

```
const book = [
    "Twinkle, twinkle, little bat!",
    "How I wonder what you're at!",
    "Up above the world you fly,",
    "Like a tea tray in the sky.",
    "Twinkle, twinkle, little bat!",
    "How I wonder what you're at!",
];
```

Теперь, когда у нас есть массив `book`, мы можем получить итератор, используя его метод `values`.

```
const it = book.values();
```

Продолжая нашу аналогию, итератор (обычно сокращаемый как `it`) является закладкой, но это закладка только для конкретной книги. Кроме того, мы ее еще никуда не поместили и мы еще не начали читать книгу. Чтобы "начать читать", необходим вызов метода `next` итератора, который возвращает объект с двумя свойствами: `value`, который содержит текущую "страницу", и `done`, которому присваивается значение `true` после того, как вы прочитаете последнюю страницу. Наша книга — длиной лишь шесть страниц, поэтому довольно просто продемонстрировать, как мы можем прочитать ее полностью.

[1] Пересказ с английского Бориса Заходера. — *Примеч. ред.*

```
it.next(); // { value: "Twinkle, twinkle, little bat!", done: false }
it.next(); // { value: "How I wonder what you're at!", done: false }
it.next(); // { value: "Up above the world you fly,", done: false }
it.next(); // { value: "Like a tea tray in the sky.", done: false }
it.next(); // { value: "Twinkle, twinkle, little bat!", done: false }
it.next(); // { value: "How I wonder what you're at!", done: false }
it.next(); // { value: undefined, done: true }
it.next(); // { value: undefined, done: true }
it.next(); // { value: undefined, done: true }
```

Здесь есть несколько важных моментов, на которые стоит обратить внимание. Прежде всего, когда метод `next` возвращает последнюю страницу книги, он не указывает, что это конец. Здесь аналогия с книгой немного нарушается: читая последнюю страницу книги, вы знаете, что она последняя, правильно? Итераторы применяются не для книг, и не всегда настолько просто узнать, когда вы закончили. Обратите внимание, что *по завершении* свойство `value` имеет значение `undefined` и вы можете продолжать вызывать метод `next` и он будет продолжать возвращать то же самое. Как только итератор достигает конца, он в буквальном смысле достигает конца набора данных, и больше не должен возвращать никаких данных.[2]

Хотя в этом примере и не показано все непосредственно, вам уже должно быть понятно, что *между* вызовами метода `it.next()` можно выполнять некие действия. Другими словами, итератор `it` всегда хранит указатель на текущее место.

Если нужно перебрать массив, можно использовать цикл `for` или цикл `for...of`. Механика цикла `for` проста: известно, что элементы массива пронумерованы и последовательны, поэтому мы можем использовать индексную переменную для доступа к каждому элементу массива по очереди. А как насчет цикла `for...of`? Как он делает свою работу без индекса? Оказывается он использует итератор: цикл `for...of` будет работать с *любым* объектом, который предоставляет итератор. Мы скоро увидим, как использовать это в своих интересах. Сначала давайте рассмотрим, как мы можем сэмулировать цикл `for...of`, используя цикл `while` и наше обретенное знание итераторов.

```
const it = book.values();
let current = it.next();
while(!current.done) {
    console.log(current.value);
    current = it.next();
}
```

Обратите внимание, что итераторы индивидуальны, т.е. каждый раз, создавая новый итератор, вы начинаете с начала и вполне можете получить несколько итераторов, которые отмечают разные места.

[2] Поскольку объекты сами отвечают за предоставление собственного итеративного механизма, как мы увидим вскоре, фактически вполне возможно создать "плохой итератор", в котором значение свойства `done` изменено на обратное; такой итератор считался бы дефектным. Вообще, вы должны полагаться на правильное поведение итератора.

```
const it1 = book.values();
const it2 = book.values();
// оба итератора в начальном положении

// чтение двух страниц с it1:
it1.next(); // { value: "Twinkle, twinkle, little bat!", done: false }
it1.next(); // { value: "How I wonder what you're at!", done: false }

// чтение одной страницы с it2:
it2.next(); // { value: "Twinkle, twinkle, little bat!", done: false }

// чтение другой страницы с it1:
it1.next(); // { value: "Up above the world you fly,", done: false }
```

В данном примере эти два итератора независимы и перебирают массив по собственному индивидуальному расписанию.

Протокол итератора

Итераторы сами по себе мало интересны: они — лишь инструмент, обеспечивающий более интересные действия. *Протокол итератора* (iterator protocol) позволяет стать итерируемым любому объекту. Предположим, что вы хотите создать класс системного журнала, в котором сообщениям будут добавляться временные метки. Внутренне для хранения сообщений с временными метками мы используем массив.

```
class Log {
    constructor() {
        this.messages = [];
    }
    add(message) {
        this.messages.push({ message: message, timestamp: Date.now() });
    }
}
```

Пока неплохо... Но что если мы захотим впоследствии перебрать все элементы в журнале (т.е. выполнить их итерацию)? Конечно, мы могли бы обращаться напрямую к массиву `log.messages`, но было бы куда лучше, если бы мы могли обработать `log` так, как будто он непосредственно итерируем, подобно массиву? Протокол итератора позволяет нам сделать это. Он гласит, что если ваш класс предоставляет символьный метод `Symbol.iterator`, который возвращает объект с поведением итератора (т.е. у него есть метод `next`, возвращающий объект со свойствами `value` и `done`), то он итерируем! Давайте изменим наш класс `Log` так, чтобы он имел метод `Symbol.iterator`.

```
class Log {
    constructor() {
        this.messages = [];
    }
    add(message) {
```

```
        this.messages.push({ message: message, timestamp: Date.now() });
    }
     [Symbol.iterator]() {
          return this.messages.values();
     }
}
```

Теперь мы можем перебрать содержимое экземпляра класса `Log` точно так же, как если бы это был массив.

```
const log = new Log();
log.add("Первый день на море");
log.add("Видели большую рыбу");
log.add("Видели корабль");
//...

// перебор log, как будто это массив!
for(let entry of log) {
    console.log('${entry.message} @ ${entry.timestamp}');
}
```

В этом примере мы соблюдаем протокол итератора, получив итератор массива `messages`, но мы вполне могли бы написать и собственный итератор.

```
class Log {
    //...

     [Symbol.iterator]() {
        let i = 0;
        const messages = this.messages;
        return {
            next() {
                if(i >= messages.length)
                    return { value: undefined, done: true };
                return { value: messages[i++], done: false };
            }
        }
    }
}
```

В рассмотренных здесь примерах задействован перебор предопределенного набора элементов: страниц в книге или сообщений в журнале. Но итераторы можно использовать и для представления объекта, значения которого никогда не исчерпаются.

Для демонстрации рассмотрим очень простой пример: создание чисел Фибоначчи. Числа Фибоначчи не особенно трудно создавать, но они зависят от предыдущих чисел. Для непосвященного: последовательность Фибоначчи — это сумма предыдущих двух чисел в последовательности. Последовательность начинается с 1 и 1: следующее число 1 + 1 равно 2. Следующее число 1 + 2 равно 3. Четвертое число 2 + 3 равно 5 и т.д. Последовательность выглядит следующим образом.

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,...
```

Последовательность Фибоначчи продолжается бесконечно, и наше приложение не знает, сколько элементов будет необходимо, что делает ее идеальным применением для итераторов. Единственное различие между этим и предыдущими примерами в том, что этот итератор никогда не будет возвращать `true` для свойства `done`:

```
class FibonacciSequence {
     [Symbol.iterator]() {
        let a = 0, b = 1;
        return {
            next() {
                let rval = { value: b, done: false };
                b += a;
                a = rval.value;
                return rval;
            }
        };
    }
}
```

Если бы мы использовали экземпляр `FibonacciSequence` с циклом `for...of`, то получили бы бесконечный цикл... ведь числа Фибоначчи никогда не закончатся, никогда! Чтобы предотвратить это, мы добавим оператор `break` после 10 элементов.

```
const fib = new FibonacciSequence();
let i = 0;
for(let n of fib) {
    console.log(n);
    if(++i > 9) break;
}
```

Генераторы

Генераторы (generator) — это функции, которые используют итератор для контроля своего выполнения. Обычная функция получает аргументы и возвращает значение, но никакого контроля вызывающая сторона над ней не имеет. Когда вы вызываете функцию, управление остается у нее до завершения. Генераторы, напротив, позволяют вам контролировать исполнение функции.

Генераторы приносят двоякую пользу. Во-первых, они позволяют контролировать выполнение функции, разделяя ее на дискретные этапы. Во-вторых, они позволяют взаимодействовать с функцией по мере ее выполнения.

Генератор похож на обычную функцию за двумя исключениями.

- Функция может *возвратить* (yield) управление вызывающей стороне в любой момент.
- Когда вы вызываете генератор, он не запускается сразу. Вместо этого вы получаете итератор. Функция запускается при вызове метода `next` итератора.

Генераторы в JavaScript отмечаются звездочками после ключевого слова `function`; в остальном их синтаксис идентичен обычным функциям. Если функция является генератором, вы можете использовать ключевое слово `yield` в дополнение к `return`.

Давайте рассмотрим простой пример генератора, который возвращает все цвета радуги.

```
function* rainbow() { // звездочка указывает, что это генератор
    yield 'красный';
    yield 'оранжевый';
    yield 'желтый';
    yield 'зеленый';
    yield 'голубой';
    yield 'синий';
    yield 'фиолетовый';
}
```

Теперь давайте рассмотрим, как происходит вызов этого генератора. Помните, что при вызове генератора возвращается итератор. Мы вызовем функцию, а затем пролистаем итератор.

```
const it = rainbow();
it.next();  // { value: "красный", done: false }
it.next();  // { value: "оранжевый", done: false }
it.next();  // { value: "желтый", done: false }
it.next();  // { value: "зеленый", done: false }
it.next();  // { value: "голубой", done: false }
it.next();  // { value: "синий", done: false }
it.next();  // { value: "фиолетовый", done: false }
it.next();  // { value: undefined, done: true }
```

Поскольку генератор `rainbow` возвращает итератор, мы можем также использовать его в цикле `for...of`.

```
for(let color of rainbow()) {
    console.log(color);
}
```

Это выведет все цвета радуги!

Выражения `yield` и двухсторонняя связь

Как уже упоминалось, генераторы обеспечивают *двухстороннюю* связь между генератором и его вызывающей стороной. Для этого используется выражение `yield`. Помните, что выражения возвращают значение, и выражение `yield` тоже должно что-то возвращать. Оно возвращает аргументы (если они есть), предоставленные вызывающей стороной при каждом вызове метода `next` итератора генератора. Давайте рассмотрим генератор, который способен поддерживать диалог.

```
function* interrogate() {
    const name = yield "Как вас зовут?";
```

```
    const color = yield "Какой ваш любимый цвет?";
    return 'У ${name} любимый цвет ${color}.';
}
```

Вызывая этот генератор, мы получаем итератор, но никакая часть генератора еще не была выполнена. Когда происходит вызов метода `next`, он пытается выполнить первую строку. Но поскольку в этой строке содержится выражение `yield`, генератор должен возвратить управление вызывающей стороне. Вызывающая сторона должна снова вызвать метод `next`, прежде чем первая строка будет выполнена и переменной `name` будет присвоено значение, которое было передано в `next`. Вот как это будет выглядеть, когда мы запустим этот генератор до конца.

```
const it = interrogate();
it.next();          // { value: "Как вас зовут?", done: false }
it.next('Коля');  // { value: "Какой ваш любимый цвет?", done: false }
it.next('оранжевый'); // { value: "У Коля любимый цвет оранжевый.", done: true }
```

Последовательность событий при выполнении этого генератора представлена на рис. 12.1.

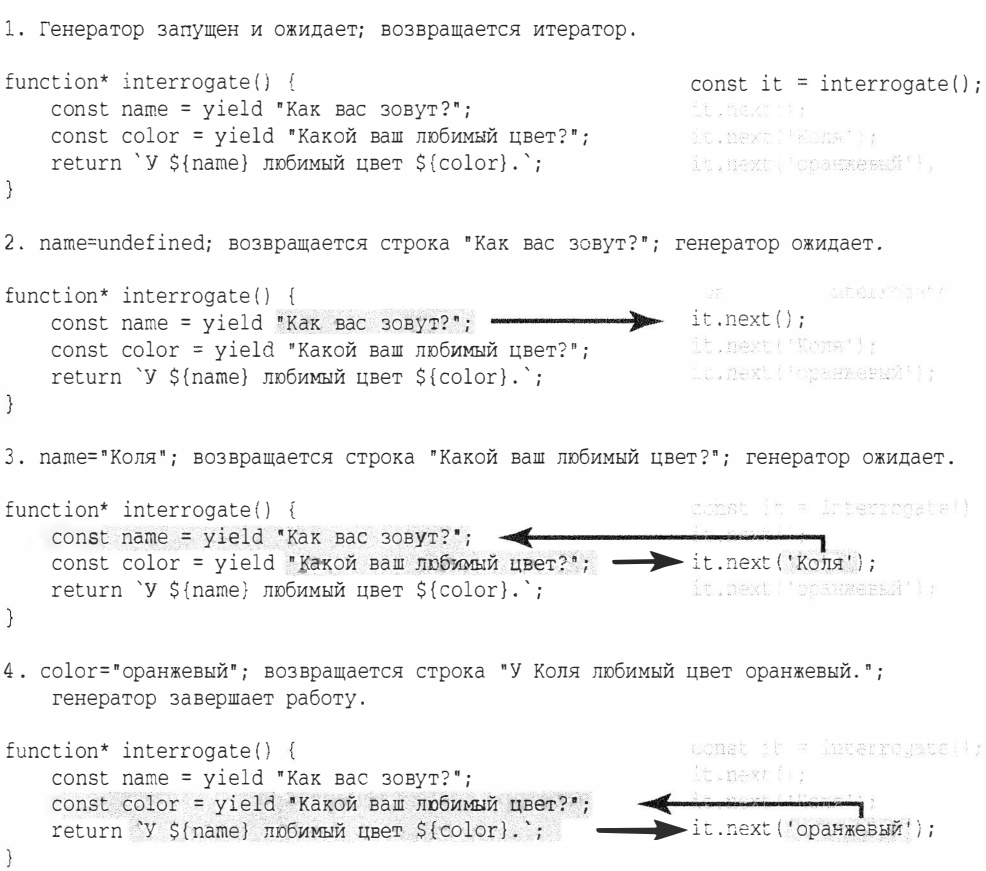


Рис. 12.1. Пример работы генератора

В этом примере показано, что генераторы очень мощны; они позволяют вызывающей стороне контролировать выполнение функций. Кроме того, поскольку вызывающая сторона может передать информацию в генератор, он может даже изменить свое поведение на основании переданной информации.

Вы не можете создать генератор, используя стрелочную нотацию; для этого следует использовать ключевое слово `function*`.

Генераторы и оператор `return`

Оператор `yield` сам по себе не завершает генератор, даже если это его последний оператор. Вызов оператора `return` в любом месте генератора приводит к присвоению значения `true` свойству `done`. При этом не имеет значения, что вы возвращаете, как показано ниже.

```
function* abc() {
    yield 'a';
    yield 'b';
    return 'c';
}

const it = count();
it.next();    // { value: 'a', done: false }
it.next();    // { value: 'b', done: false }
it.next();    // { value: 'c', done: true }
```

Хотя это и вполне корректное поведение генератора, имейте в виду, что при использовании генераторов не всегда обращают внимание на свойство `value`, когда `done` — `true`. Например, если мы будем использовать это в цикле `for...of`, "c" не будет выведено вообще.

```
// выведет "a" и "b", но не "c"
for(let l of abc()) {
    console.log(l);
}
```

Я не рекомендую использовать оператор `return` для возвращения важного значения из генератора. Для этого лучше использовать оператор `yield`; а оператор `return` лучше использовать только для экстренной остановки генератора. Поэтому я рекомендую вообще не указывать значений в операторе `return` генератора.

Заключение

Итераторы обеспечивают стандартный механизм для коллекций или объектов, способных содержать несколько значений. Хотя итераторы не предоставляют ничего такого, что нельзя было бы реализовать до появления ES6, они стандартизируют действительно важное и общее действие.

Генераторы позволяют создавать намного более управляемые и изменяющие свое поведение функции: вызывающая сторона больше не должна заранее предоставлять данные функции, ожидать завершения ее выполнения и получения результата. По существу, генераторы позволяют отсрочить вычисления и выполнять их только по мере необходимости. В главе 14 мы увидим, как они позволяют реализовать мощные схемы для управления асинхронным кодом.

ГЛАВА 13

Функции и мощь абстрактного мышления

Если бы JavaScript был Бродвейской постановкой, то функции были бы блестящей звездой: они постоянно купались бы в свете софитов и выходили бы на поклон под гром аплодисментов (а иногда, без сомненья, и под свист: всем не понравишься). Мы рассмотрели механизм функций в главе 6, а здесь рассмотрим способы применения функций и то, как они могут преобразить ваш подход к решению задач.

Сама концепция функций подобна хамелеону: в различных контекстах они проявляют свои разные стороны. Первый (и самый простой) аспект функций — многократное использование кода.

Функции как подпрограммы

Идея *подпрограмм* (subroutine) — очень старый практический подход для снижения сложности. Без подпрограмм программирование было бы весьма монотонным делом. Подпрограммы просто упаковывают некий объем повторяемых функциональных возможностей, присваивают им имя и позволяют их выполнять в любое время, обратившись по этому имени.

Подпрограммы известны также под названиями *процедура* (procedure), *функция* (routine), *подпрограмма* (subprogram), *макрос* (macro) и очень расплывчатым и обобщенным *вызываемый блок* (callable unit). Обратите внимание, что в JavaScript мы фактически не используем слово *подпрограмма* (subroutine). Мы просто называем функцию функцией (или методом). Здесь мы употребляем термин *подпрограмма*, только чтобы подчеркнуть столь простой способ использования функций.

Довольно часто подпрограмма используется для упаковки *алгоритма*, что является простым и понятым способом решения данной задачи. Давайте рассмотрим алгоритм определения, принадлежит ли текущая дата високосном году.

```
const year = new Date().getFullYear();
if(year % 4 !== 0) console.log('${year} не високосный.')
else if(year % 100 != 0) console.log('${year} високосный.')
else if(year % 400 != 0) console.log('${year} не високосный.')
else console.log('{$year} високосный ');
```

Вообразите, что вы должны выполнить этот код в программе 10 или даже 100 раз. Теперь предположим, что понадобилось изменить формулировку сообщения, которое выводится на консоль; вам придется найти все случаи использования этого кода и в каждом изменить по четыре строки! Это именно та проблема, которую решают подпрограммы. В JavaScript эту потребность может удовлетворить функция.

```
function printLeapYearStatus() {
    const year = new Date().getFullYear();
    if(year % 4 !== 0) console.log('${year} не високосный.')
    else if(year % 100 != 0) console.log('${year} високосный.')
    else if(year % 400 != 0) console.log('${year} не високосный.')
    else console.log('{$year} високосный.');
}
```

Мы создали *многократно используемую* подпрограмму (функцию) `printLeapYearStatus`. Теперь это должно быть вам вполне знакомо.

Обратите внимание на имя, которое мы выбрали для функции: `printLeapYearStatus`. Почему не `getLeapYearStatus` или `leapYearStatus`, или просто `leapYear`? Хотя эти имена были бы короче, они упускают важную деталь: эта функция просто выводит текущее состояние високосного года. Осмысленное имя функции — это отчасти наука, отчасти искусство. Имя — не для JavaScript, его не заботит, какое имя вы используете. Имя — для людей (или для вас в будущем). Когда вы называете функции, думайте о том, что представит себе человек, если будет судить о функции только по ее имени. В идеале имя должно точно сообщать, что делает функция. С другой стороны, имя функции может быть *слишком подробным*. Например, мы могли бы назвать эту функцию `calculateCurrentLeapYearStatusAndPrintToConsole`, но дополнительная информация в таком длинном имени явно зашкаливает. Вот где начинается искусство.

Функции как подпрограммы, возвращающие значение

Функция `printLeapYearStatus` в нашем предыдущем примере — это подпрограмма в обычном смысле слова: она лишь объединяет несколько функциональных возможностей для удобства многократного использования, не более. Это простейшее использование функций, к которому вы будете прибегать не очень часто, и даже еще реже, когда ваш подход к программированию станет более сложным и абстрактным. Давайте сделаем следующий шаг в направлении абстрактного мышления и рассмотрим функции как подпрограммы, которые *возвращают значение*.

Функция `printLeapYearStatus` хороша, но, когда мы начнем создавать свои программы, простого вывода на консоль часто становится недостаточно. Теперь мы хотим использовать для вывода HTML-код или выполнять запись в файл, или использовать текущее состояние високосного года в *других* вычислениях. Но мы все еще не хотим возвращаться к обстоятельному объяснению нашего алгоритма каждый раз, когда хотим знать, является ли текущий год високосным.

К счастью, нашу функцию достаточно просто переписать (и переименовать!) так, чтобы она стала подпрограммой, которая возвращает значение.

```
function isCurrentYearLeapYear() {
    const year = new Date().getFullYear();
    if(year % 4 !== 0) return false;
    else if(year % 100 != 0) return true;
    else if(year % 400 != 0) return false;
    else return true;
}
```

Теперь давайте рассмотрим некоторые примеры того, как мы могли бы использовать возвращаемое значение нашей новой функции.

```
const daysInMonth =
    [31, isCurrentYearLeapYear() ? 29 : 28, 31, 30, 31, 30,
        31, 31, 30, 31, 30, 31];
if(isCurrentYearLeapYear()) console.log('Сейчас високосный год.');
```

Прежде чем двигаться дальше, давайте рассмотрим почему мы выбрали для этой функции именно такое название. Весьма популярно начинать имена функций, которые возвращают логическое значение (или предназначены для использования в логическом контексте), с сочетания букв `is`. Мы также включали в имя функции слово *current* (текущее). Почему? Потому что в этой функции *текущая дата* используется явно. Другими словами, эта функция возвратит разные значения, если вы запустите ее 31 декабря 2016 года, а затем на следующий день — 1 января 2017 года.

Функции как... функции

Теперь, когда мы рассмотрели некоторые из наиболее очевидных способов использования функций, пришло время подумать о функциях как... о функциях. Если бы вы были математиком, то вы думали бы о функции как о *зависимости* (relation) выходных данных от входных. Любой вывод зависит от ввода. Программисты считают функции, которые придерживаются математического определения, *чистыми функциями* (pure function). В некоторых языках (таких, как Haskell) допускаются только чистые функции.

Чем такая функция отличается от функций, которые мы уже рассматривали? Во-первых, чистая функция должна *всегда возвращать одно и то же значение для одного и того же набора входных данных*. Функция `isCurrentYearLeapYear` не является

чистой, поскольку возвращает то одно значение, то другое в зависимости от того, когда вы ее вызовете (в один год она может возвратить `true`, а на следующий — `false`). Во-вторых, у функции не должно быть *побочных эффектов* (side effect). Таким образом, вызов функции не должен изменять состояние программы. В нашем обсуждении мы еще не встречали функций с побочными эффектами (мы не считаем вывод на консоль побочным эффектом). Давайте рассмотрим простой пример.

```
const colors = ['красный', 'оранжевый', 'желтый', 'зеленый',
                'голубой', 'синий', 'фиолетовый'];
let colorIndex = -1;
function getNextRainbowColor() {
    if(++colorIndex >= colors.length) colorIndex = 0;
    return colors[colorIndex];
}
```

Функция `getNextRainbowColor` возвращает каждый раз другой цвет, циклически проходя все цвета радуги. Эта функция нарушает оба правила чистой функции: у нее разные возвращаемые значение для одного и того же входного значения (у нее нет аргументов, поэтому ее входного значение — ничего), и ее вызов приводит к побочному эффекту (изменение значения переменной `colorIndex`). Переменная `colorIndex` не является частью функции; вызов `getNextRainbowColor` модифицирует ее, что является побочным эффектом.

Вернемся на мгновение к нашей задаче определения високосного года. Как мы можем преобразовать свою функцию високосного года в чистую функцию? Просто!

```
function isLeapYear(year) {
    if(year % 4 !== 0) return false;
    else if(year % 100 != 0) return true;
    else if(year % 400 != 0) return false;
    else return true;
}
```

Эта новая функция всегда будет возвращать одно и то же значение для одного и того же входного значения, и она не вызывает побочных эффектов, что делает ее чистой.

Наша функция `getNextRainbowColor` немного сложнее. Мы можем устранить побочный эффект, поместив внешние переменные в замкнутое выражение.

```
const getNextRainbowColor = (function() {
    const colors = ['красный', 'оранжевый', 'желтый', 'зеленый',
                    'голубой', 'синий', 'фиолетовый'];
    let colorIndex = -1;
    return function() {
        if(++colorIndex >= colors.length) colorIndex = 0;
        return colors[colorIndex];
    };
})();
```

Теперь у нас есть функция без побочных эффектов, но это все еще не чистая функция, поскольку она не всегда возвращает один и тот же результат для одного и того же ввода. Для устранения этой проблемы следует тщательно рассмотреть, как мы используем данную функцию. Есть шанс, что мы будем вызывать ее циклически, например в браузере, чтобы изменять цвет элемента два раза в секунду (о коде для браузера мы узнаем больше в главе 18).

```
setInterval(function() {
    document.querySelector('.rainbow')
        .style['background-color'] = getNextRainbowColor();
}, 500);
```

Выглядит не так уж и плохо, и, конечно, намерение вполне однозначно: некий элемент HTML использует класс `rainbow` для циклической смены цвета. Проблема в том, что если что-то еще вызовет метод `getNextRainbowColor()`, то оно вмешается в работу этого кода! Здесь стоит остановиться и задаться вопросом "Настолько ли хорошей идеей является функция с побочными эффектами". В данном случае, вероятно, лучшим выбором был бы итератор.

```
function getRainbowIterator() {
    const colors = ['красный', 'оранжевый', 'желтый', 'зеленый',
                    'голубой', 'синий', 'фиолетовый'];
    let colorIndex = -1;
    return {
        next() {
            if(++colorIndex >= colors.length) colorIndex = 0;
            return { value: colors[colorIndex], done: false };
        }
    };
}
```

Наша функция `getRainbowIterator` является теперь чистой: она возвращает одно и то же каждый раз (итератор), и у нее нет никаких побочных эффектов. Мы могли бы использовать это иначе, но так намного безопаснее.

```
const rainbowIterator = getRainbowIterator();
setInterval(function() {
    document.querySelector('.rainbow')
        .style['background-color'] = rainbowIterator.next().value;
}, 500);
```

Вы могли бы подумать, что это только поверхностное решение проблемы: разве метод `next()` не возвращает разные значения каждый раз? Так и есть, однако вспомните, что `next()` является *методом*, а не функцией. Он работает в контексте объекта, которому принадлежит, поэтому его поведение контролируется этим объектом. Если мы будем использовать `getRainbowIterator` в других частях нашей программы, то они создадут разные итераторы, которые не будут конфликтовать ни с какими другими итераторами.

И что?

Теперь, увидев три разные шляпы, которые может носить функция (подпрограмма, подпрограмма с возвращаемым значением и чистая функция), мы сделаем паузу и спросим себя "И что?[1] Почему эти различия имеют значение?"

Моя задача в этой главе не столько объяснить синтаксис JavaScript, как заставить вас думать, *почему именно* это так. Зачем нужны функции? Рассматривая функции как подпрограммы, можно найти один из ответов на этот вопрос: чтобы избежать повторов. Подпрограммы позволяют упаковывать общепринятые функциональные возможности — довольно очевидное преимущество.

Избежание повторения кода за счет упаковки является настолько основополагающей концепцией, что для нее есть собственная аббревиатура: DRY (don't repeat yourself — *не повторяйся*). Хотя она, возможно, лингвистически и сомнительна (дословно — "сухой"), вы найдете, что, описывая код, люди используют эту аббревиатуру как прилагательное. "Этот код мог бы быть более СУХИМ". Если кто-то говорит вам это, то имеет в виду, что вы излишне повторяете функциональные возможности.

С чистыми функциями дела обстоят несколько хуже — они отвечают на вопрос "Почему" немного более абстрактным способом. Один из ответов мог бы быть таким: "Потому что они делают программирование более похожим на математику!" Это ответ, который мог бы вызвать следующий вопрос: "И почему это хорошо?" Наилучший ответ мог бы быть таким: "Потому что чистые функции делают код проще и понятнее, облегчают его проверку и делают более переносимым".

Функции, которые возвращают разные значения при разных обстоятельствах или имеют побочные эффекты, *привязаны к своему контексту*. Если у вас есть действительно полезная функция с побочными эффектами, например, и вы извлекаете ее из одной программы, чтобы поместить в другую, это может не сработать. Или, что хуже того, она может сработать в 99% случаев, а в 1% привести к серьезной ошибке. Любой программист знает, что неустойчивые ошибки — самый плохой вид ошибок: они могут долго оставаться незамеченными, а когда обнаруживаются, поиск причин их возникновения напоминает поиск иголки в стоге сена.

Если чистые функции лучше всех, то напрашивается вполне резонный вывод, что *вы всегда должны предпочитать чистые функции*. Я говорю "предпочитать" поскольку иногда проще создать функцию с побочными эффектами. Начинающие программисты испытывают желание делать это весьма часто. Я не собираюсь отговаривать вас от этого, я просто рекомендую вам остановиться и подумать, можете ли вы

[1] Композиция "So What" была исполнена Pink в 2008 году. — *Примеч. ред.*

найти способ использовать чистую функцию вместо нее. Со временем вы, естественно, будете стремиться к чистым функциям.

Объектно-ориентированное программирование, которое мы рассматривали в главе 9, предоставляет парадигму, позволяющую использовать побочные эффекты контролированным и разумным способом за счет ограничения их области видимости.

Функции являются объектами

В JavaScript функции являются экземплярами объекта `Function`. С практической точки зрения это никак не влияет на то, как вы их используете; это просто информация к размышлению. Заслуживает внимания то, что, если вы попытаетесь идентифицировать тип переменной `v`, оператор `typeof v` возвратит для функций `"function"`. Это весьма разумно, в отличие от случая, когда `v` является массивом: он возвратит `"object"`. В результате вы можете использовать оператор `typeof v` для идентификации функции. Обратите, однако, внимание, что если `v` — это функция, то `v instanceof Object` даст истину. Поэтому, если вам нужно отличать функции от объектов других типов, то для проверки сначала используйте оператор `typeof`.

Немедленно вызываемое функциональное выражение и асинхронный код

Мы познакомились с немедленно вызываемыми функциональными выражениями (IIFE) в главе 6 и увидели, что они позволяют создавать замкнутые выражения. Давайте рассмотрим важный пример (к которому мы вернемся в главе 14) того, как IIFE может помочь нам с асинхронным кодом.

Один из первых случаев использования IIFE подразумевает создание новых переменных в новых областях видимости, чтобы асинхронный код выполнялся правильно. Рассмотрим классический пример таймера, который осуществляет обратный отсчет от 5 секунд до 0 (команда `"Старт!"`). В этом коде использована встроенная функция `setTimeout`, которая задерживает выполнение ее первого аргумента (функции) на второй аргумент (количество миллисекунд). Например, следующий код выводит строку `"Привет!"` через 1,5 секунды.

```
setTimeout(function() { console.log("Привет!"); }, 1500);
```

Теперь, обладая этим знанием, создадим нашу функцию обратного отсчета.

```
var i;
for(i=5; i>=0; i--) {
    setTimeout(function() {
        console.log(i===0 ? "Старт!" : i);
    }, (5-i)*1000);
}
```

Обратите внимание, что здесь мы используем `var` вместо `let`. Давайте рассмотрим, почему IIFE так важны. Если вы ожидаете увидеть вывод `5`, `4`, `3`, `2`, `1`, `"Старт!"`, то будете разочарованы. Вместо этого вы найдете шесть раз выведенное число `-1`. Дело в том, что функция, передаваемая `setTimeout`, вызывается не сразу в цикле, а через некоторое время. Таким образом, цикл выполнится начиная с `i`, равного `5`, и в конечном счете достигнет `-1`... еще до того, как любая из функций будет вызвана. Таким образом, на момент вызова функции `i` будет иметь значение `-1`.

Даже при том, что область видимости уровня блока (с переменными `let`), по существу, решает эту проблему, данный пример все еще очень важен, если вы новичок в асинхронном программировании. Трудно объять необъятное, но понимание асинхронного выполнения критически важно (глава 14).

Без применения переменных области видимости уровня блока для решения этой задачи нужно было использовать другую функцию. При использовании дополнительной функции создается новая область видимости, и значение `i` может быть "зафиксировано" (в замкнутом выражении) на каждом этапе. Рассмотрим сначала использование именованной функции.

```
function loopBody(i) {
    setTimeout(function() {
        console.log(i===0 ? "Старт!" : i);
    }, (5-i)*1000);
}
var i;
for(i=5; i>0; i--) {
    loopBody(i);
}
```

На каждом этапе цикла вызывается функция `loopBody`. Напомню, что в JavaScript аргументы передаются функции при вызове по значению. Таким образом, на каждом этапе функции передается не переменная `i`, а ее *значение*. Вначале передается значение `5`, во второй раз — значение `4` и т.д. Не имеет значения, что в обоих местах мы используем имя переменной (`i`): по существу, мы создаем шесть разных областей видимости и шесть независимых переменных (одну для внешней области видимости и пять для каждого из вызовов `loopBody`).

Но все же создание именованной функции для цикла, который вы собираетесь использовать только однажды, довольно утомительное занятие. Задействуйте IIFE: они, по существу, создают эквивалентные анонимные функции, которые вызываются немедленно. Вот как предыдущий пример выглядит с IIFE.

```
var i;
for(i=5; i>0; i--) {
    (function(i) {
        setTimeout(function() {
            console.log(i===0 ? "Старт!" : i);
```

```
        }, (5-i)*1000);
    })(i);
}
```

Как много скобок! Если проанализировать их, то можно заметить, что здесь происходит то же самое: мы создаем функцию, которой передается один аргумент, и она вызывается на каждом шаге цикла (рис. 13.1).

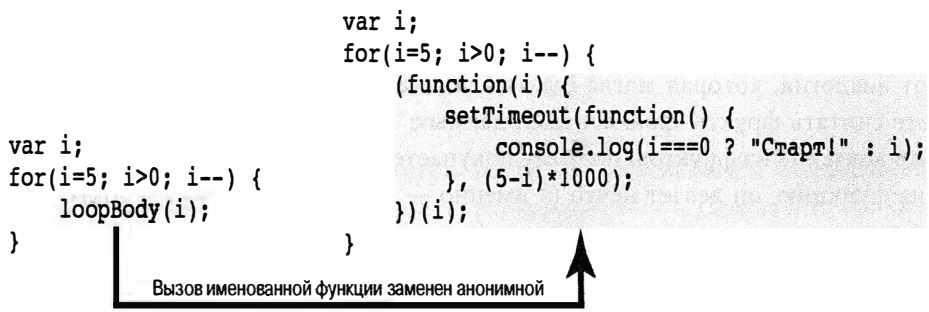


Рис. 13.1. Немедленно вызываемое функциональное выражение

Переменные области видимости блока решают эту задачу без введения дополнительной функции, чтобы создать новую область видимости. Использование переменных области видимости блока существенно упрощает этот пример.

```
for(let i=5; i>0; i--) {
    setTimeout(function() {
        console.log(i===0 ? "Старт!" : i);
    }, (5-i)*1000);
}
```

Обратите внимание, что мы используем ключевое слово `let` *в аргументах* цикла `for`. Если бы мы поместили его вне цикла `for`, у нас была бы та же проблема, что и прежде. Таким образом, использование ключевого слова `let` сообщает JavaScript, что на каждом этапе цикла должна быть новая, независимая копия переменной `i`. В результате, когда функции, переданные `setTimeout`, выполняются в будущем, они каждый раз получают свое значение переменной в ее собственной области видимости.

Переменные функций

Если вы новичок в программировании, можете налить себе еще кофе[2] и усесться поудобнее: в этом разделе рассматривается концепция, которая очень важна, но у новичков зачастую вызывает затруднения.

[2] Имеется в виду композиция "Another Cup of Coffee" группы Mike And The Mechanics. — *Примеч. ред.*

Проще всего считать числа, строки и даже массивы обычными переменными. Эта концепция приводит нас к очень удобной идее, что переменная — это фрагмент данных (набор, или коллекция данных, в случае массива или объекта). Если размышлять именно так о переменных, то будет куда сложнее понять и реализовать весь потенциал функций, поскольку вы можете передавать функцию в виде параметра другой функции точно так же, как если бы это была любая другая переменная. Поскольку функции *активны*, мы не думаем о них как о фрагментах данных (которые мы считаем пассивными). И это правда, функция активна, когда ее *вызывают*. Однако до вызова… она пассивна, точно так же, как любая другая переменная.

Вот аналогия, которая могла бы вам помочь. Если вы идете в супермаркет, то можете считать фрукты фрагментами данных: 2 банана, 1 яблоко и т.д. Вы хотите сделать коктейль из фруктов, поэтому покупаете еще и блендер. Блендер больше похож на функцию: он делает нечто (а именно — измельчает фрукты в коктейль). Но пока блендер находится в вашей корзине (выключенный и ничего не смешивающий), это только еще один товар в вашей корзине, и вы можете сделать с ним то же, что и с фруктами: переложить из корзины на кассовую ленту, заплатить за него, положить в хозяйственную сумку, отнести домой и т.д. И только когда вы подключите его к сети, загрузите фруктами и включите, он станет чем-то отличным от фруктов.

Таким образом, функция применима везде, где применима переменная, но что это значит? В частности, это значит, что вы можете делать следующие вещи.

- Присвоить функции псевдоним, что создаст переменную, которая указывает на нее.
- Поместить функцию в массив (возможно, смешанный, из данных других типов).
- Использовать функцию как свойство объекта (см. главу 9).
- Передать функцию в функцию.
- Возвратить функцию из функции.
- Возвратить функцию из функции, которой в качестве аргумента была передана функция.

Голова еще не кружится? Этот список действительно кажется очень абстрактным, и вы, естественно, могли бы задаться вопросом “С какой стати я должен это использовать?” На самом деле эта гибкость невероятно мощна, и все эти возможности применяются весьма часто.

Давайте начнем с самого простого элемента этого списка: присвоения функции псевдонима. Предположим, что у вас есть функция с очень длинным именем и вы хотите использовать ее многократно в пределах нескольких строк. Вы быстро устанете вводить такое длинное название функции, да и читать код будет очень трудно. Поскольку функция — это только тип данных, как и любой другой, вы можете создать новую переменную с более коротким именем.

```
function addThreeSquareAddFiveTakeSquareRoot(x) {
    // это совсем бестолковая функция, не так ли?
    return Math.sqrt(Math.pow(x+3, 2)+5);
}

// до
const answer = (addThreeSquareAddFiveTakeSquareRoot(5) +
    addThreeSquareAddFiveTakeSquareRoot(2)) /
    addThreeSquareAddFiveTakeSqureRoot(7);

// после
const f = addThreeSquareAddFiveTakeSquareRoot;
const answer = (f(5) + f(2)) / f(7);
```

Обратите внимание, что в примере "после" мы *не используем круглые скобки после* имени функции `addThreeSquareAddFiveTakeSquareRoot`. Сделав это, мы *вызвали бы* функцию, и переменная `f` вместо того, чтобы стать псевдонимом `addThreeSquareAddFiveTakeSquareRoot`, содержала бы *результат* этого вызова. Затем, когда мы попытались бы использовать ее как функцию (например, `f(5)`), это привело бы к ошибке, поскольку `f` не была бы функцией, а вызывать вы можете только функции.

Конечно, это совершенно надуманный пример, и в действительности встречается не часто. Но это на самом деле происходит при *применении пространств имен* (namespacing), что весьма распространено при разработке приложений для Node (см. главу 20), например так.

```
const Money = require('math-money'); // require - функция Node для
                                     // импорта библиотек
const oneDollar = Money.Dollar(1);
// или, если мы не хотим писать повсюду "Money.Dollar":
const Dollar = Money.Dollar;
const twoDollars = Dollar(2);
// обратите внимание: oneDollar и twoDollars - экземпляры того же типа
```

В данном случае эффект от *применения псевдонимов* (aliasing) не так уж и велик, `Money.Dollar` сокращается до просто `Dollar`, что кажется достаточно разумным.

Теперь, завершив умственную разминку, давайте перейдем к более энергичному абстрактному размышлению.

Функции в массиве

Исторически это программное решение использовалось не очень широко, но его популярность растет, а при определенных обстоятельствах оно чрезвычайно полезно. Одно из его применений — реализация идеи *конвейера* (pipeline), когда есть набор индивидуальных этапов, которые мы хотим выполнять часто. Преимущество использования массивов в том, что вы можете изменять их в любое время. Необходимо

удалить этапы? Достаточно удалить их из массива. Необходимо добавить этап? Достаточно добавить его в массив.

Один из примеров — графические преобразования. Если вы создаете некое программное обеспечение для визуализации, "конвейер" преобразований будет часто использоваться во многих местах. Вот пример обычных двумерных преобразований.

```
const sin = Math.sin;
const cos = Math.cos;
const theta = Math.PI/4;
const zoom = 2;
const offset = [1, -3];

const pipeline = [
    function rotate(p) {
        return {
            x: p.x * cos(theta) - p.y * sin(theta),
            y: p.x * sin(theta) + p.y * cos(theta),
        };
    },
    function scale(p) {
        return { x: p.x * zoom, y: p.y * zoom };
    },
    function translate(p) {
        return { x: p.x + offset[0], y: p.y + offset[1]; };
    },
];

// pipeline - это массив функций для определенного двумерного
// преобразования, теперь мы можем преобразовать точку:

const p = { x: 1, y: 1 };
let p2 = p;
for(let i=0; i<pipeline.length; i++) {
    p2 = pipeline[i](p2);
}

// теперь p2 - это p1, повернутая на 45 градусов (pi/4 радиан) вокруг
// начала координат, передвинутая на 2 единицы от начала координат, а
// затем смещенная на 1 единицу вправо и на 3 вниз
```

Этот очень простой пример для графических преобразований, но мы надеемся, что он дает вам представление о мощи хранения функций в массиве. Обратите внимание на синтаксис применения каждой функции в конвейере: `pipeline[i]` обращается к элементу `i` конвейера, который предоставляет функцию. Затем вызывается функция (круглые скобки). Ей передается объект точки, после чего он снова присваивается сам себе. Таким образом, объект точки — это кумулятивный результат выполнения каждого этапа в конвейере.

Конвейерная обработка нашла применение не только в прикладных программах обработки графики: она популярна и в обработке звука, и во многих научных и технических приложениях. В действительности каждый раз, когда у вас есть серия функций, которую нужно выполнять в определенном порядке, конвейер является весьма полезной абстракцией.

Передача функции в функцию

Мы уже видели несколько примеров передачи функций в функции: имеется в виду передача функций в функции `setTimeout` и `forEach`. Передача функций в функции применяется также в асинхронном программировании, которое становится все более популярным. При обычном способе достижения асинхронного выполнения подразумевается передача функций (называемых функциями *обратного вызова* (callback)) в другую функцию. Эта функция вызывается, когда содержащая функция завершает свою работу. Мы обсудим обратные вызовы подробнее в главе 14.

Обратный вызов — не единственная причина, по которой вам может понадобиться передавать функции в другую функцию; это также отличный способ "внедрить" функцию. Давайте рассмотрим функцию `sum`, которая просто суммирует все числа в массиве (для простоты мы не будем делать никакой проверки или обработки ошибок, если массив будет содержать элементы, отличные от числовых). Это достаточно простое упражнение, но что если затем понадобится функция, которая возвращает сумму *квадратов*? Конечно, мы просто могли бы написать новую функцию `sumOfSquares`..., но что будет, когда понадобится сумма кубов? Вот где возможность передать функцию может очень пригодиться. Рассмотрим эту реализацию функции `sum`.

```
function sum(arr, f) {
    // если никакой функции не предоставлено, используется "пустая
    // функция", которая просто возвращает свой аргумент неизменным
    if(typeof f != 'function') f = x => x;

    return arr.reduce((a, x) => a += f(x), 0);
}
sum([1, 2, 3]);                      // возвращает 6
sum([1, 2, 3], x => x*x);            // возвращает 14
sum([1, 2, 3], x => Math.pow(x, 3)); // возвращает 36
```

При передаче произвольной функции в `sum` мы можем заставить ее сделать... все, что хотим. Нужна сумма квадратных корней? Никаких проблем. Нужна сумма чисел, возведенных в степень 4,233? Проще простого. Обратите внимание, что мы хотим предусмотреть возможность простого вызова функции `sum`, т.е. не делая ничего специально и не передавая ей никакой функции. В функции параметр `f` имеет значение `undefined`, и если мы попытаемся его вызвать, то получим ошибку. Чтобы предотвратить это, мы превращаем нечто, не являющееся функцией, в "пустую функцию",

которая в основном ничего не делает. Таким образом, если вы передаете ей 5, она возвращает 5, и т.д. Есть более эффективные способы справиться с этой ситуацией (такой, как вызов другой функции без дополнительных затрат на вызов пустой функции для каждого элемента), но это хорошая практика, чтобы научиться создавать "безопасные" функции таким способом.

Возвращение функции из функции

Возвращение функции из функции является, вероятно, не только самым загадочным способом использования функций, но и чрезвычайно полезным. Это можно сравнить с трехмерным принтером: это вещь, которая делает нечто (как функция), что может, в свою очередь, делать нечто, что также делает что-то. Но самая захватывающая часть здесь в том, что функция, которую вы возвращаете, может быть видоизменена — очень похоже на то, как вы можете изменить то, что печатаете на трехмерном принтере.

Давайте предположим, что нашей функции `sum`, как и ранее, может передаваться (а может и нет!) функция для обработки каждого элемента массива, до того, как они будут просуммированы. Помните, как мы упоминали, что могли бы создать отдельную функцию `sumOfSquares`, если бы захотели? Давайте рассмотрим ситуацию в которой такая функция необходима, т.е. когда передавать в функцию и массив *и* другую функцию нежелательно. Нам явно нужна функция, которой передается *только* один массив значений, а она возвращает сумму его квадратов. (Если вы задаетесь вопросом, когда такое обстоятельство могло бы возникнуть, рассмотрите проект некого API, в котором вам разрешено создавать функции типа `sum`, только с одним аргументом.)

Один из подходов подразумевает создание новой функции, которая просто вызывает нашу старую функцию.

```
function sumOfSquares(arr) {
    return sum(arr, x => x*x);
}
```

Данный подход, конечно, прекрасен и вполне может сработать, если все, что необходимо, — это одна функция, но что если необходимо повторять этот шаблон раз за разом? Решением нашей проблемы могло бы быть создание функции, которая возвращает специализированную функцию.

```
function newSummer(f) {
    return arr => sum(arr, f);
}
```

Эта новая функция, `newSummer`, создает совершенно новый вариант функции `sum`, которая имеет только один аргумент, но в ней используется специальная функция. Давайте рассмотрим, как мы могли бы использовать это для получения различных видов сумм.

```
const sumOfSquares = newSummer(x => x*x);
const sumOfCubes = newSummer(x => Math.pow(x, 3));
sumOfSquares([1, 2, 3]);    // возвращает 14
sumOfCubes([1, 2, 3]);      // возвращает 36
```

Эта методика, когда мы берем функцию с несколькими аргументами и преобразуем ее в функцию с одним аргументом, называется *каррингом* (currying) в честь ее разработчика, американского математика Хаскелла Брукса Карри (Haskell Curry).

Случаи возвращения функции из функции зачастую глубоки и сложны. Если вы хотите увидеть больше примеров этого, взгляните на пакеты приложения среднего уровня для Express или Koa (популярные среды веб-разработки JavaScript).

Рекурсия

Другой весьма распространенный и важный способ использования функций — это *рекурсия* (recursion), когда функция вызывает саму себя. Это особенно мощная методика, когда функция делает то же самое с постепенно уменьшающимися наборами данных.

Давайте начнем с вымышленного примера: поиска иголки в стоге сена. Если бы у вас были реальный стог сена и игла, которую нужно найти в нем, то реальный подход мог бы быть таким.

1. Если вы можете увидеть иглу в стоге сена, перейти к п. 3.
2. Удалите часть сена из стога. Перейти к п. 1.
3. Готово!

Вы просто каждый раз уменьшаете размер стога сена, пока не находите иглу; это и есть рекурсия. Давайте посмотрим, как преобразовать этот пример в код.

```
function findNeedle(haystack) {
    if(haystack.length === 0) return "Здесь иголки нет!";
    if(haystack.shift() === 'иголка') return "Нашли!"
    return findNeedle(haystack); // стог сена уменьшился на один элемент
}

findNeedle(['сено', 'сено', 'сено', 'сено', 'иголка', 'сено', 'сено']);
```

В этой рекурсивной функции важно обратить внимание на то, что она учитывает все возможности: если массив `haystack` пуст (когда негде и ничего искать), когда иголка — первый элемент в массиве (готово!) или не первый (она находится где-то в остальной части массива, поэтому мы удаляем первый элемент и повторяем функцию; помните, что `Array.prototype.shift` удаляет первый элемент из массива по месту).

Важно, что у рекурсивной функции обязательно должно быть *условие остановки* (stopping condition); без него она продолжит вызывать саму себя до тех пор, пока интерпретатор JavaScript не решит, что стек вызовов стал слишком большим (что приведет к аварийному завершению программы). В нашей функции `findNeedle` есть два условия выхода: когда иголка найдена и когда стог закончился. Поскольку мы уменьшаем размер стога каждый раз, в конечном счете мы неизбежно достигнем одного из этих условий остановки.

Давайте рассмотрим более полезный, проверенный временем пример: поиск факториала числа. Факториал числа — это число, умноженное на все числа до него и обозначаемое восклицательным знаком после числа. Таким образом 4! вычисляется как $4 \times 3 \times 2 \times 1 = 24$. Вот как мы реализовали бы это в виде рекурсивной функции.

```
function fact(n) {
    if(n === 1) return 1;
    return n * fact(n-1);
}
```

Здесь есть условие остановки (`n === 1`), и каждый раз, делая рекурсивный вызов, мы уменьшаем значение `n` на единицу. Так мы в конечном счете доберемся до 1 (эта функция не будет правильно работать, если вы передадите ей `0` или отрицательное число, хотя, конечно, мы могли бы добавить ряд проверок, чтобы этого не случилось).

Заключение

Если у вас есть опыт работы с другими функциональными языками программирования, такими как ML, Haskell, Clojure или F#, эта глава вряд ли была для вас сложной. В противном случае она, вероятно, расширила ваш кругозор, и вы узнали немного больше об абстрактных возможностях функционального программирования (поверьте, впервые столкнувшись с этими идеями, я был, конечно, удивлен). Вы могли бы быть поражены количеством способов, которыми можно достичь той же цели, и задаться вопросом "Какой путь лучше?" Боюсь, что простого ответа на этот вопрос нет. Зачастую он зависит от решаемой задачи: некоторые из них имеют только определенную методику. Многое зависит и от вас: какие методики вам нравятся? Если представленные в этой главе методики ставят вас в тупик, то я рекомендую перечитать ее несколько раз. Изложенные здесь концепции чрезвычайно мощны, и единственный способ уяснить, какие из них будут полезны именно для вас, — это внимательно изучить их и понять.

Асинхронное программирование

Мы упоминали асинхронное программирование в главе 1, когда обсуждали взаимодействие с пользователем. Помните, что взаимодействие с пользователем, вполне естественно, является асинхронным: вы не можете контролировать, когда пользователь щелкнет, коснется, скажет или введет. Однако пользовательский ввод — не единственная причина для асинхронного выполнения программ: сама природа JavaScript вынуждает к этому во многих случаях.

Приложение JavaScript выполняется в *одном потоке* (single-threaded). Таким образом, JavaScript делает только что-то одно за один раз. Большинство современных компьютеров способно выполнять одновременно несколько операций (подразумевается, что у них есть несколько ядер), и даже компьютеры с одним ядром настолько быстры, что могут имитировать одновременное выполнение нескольких задач, выполняя сначала часть задачи A, затем — небольшую часть задачи B, затем — задачи C и так до тех пор, пока все задачи не будут выполнены (это так называемый *мультипрограммный режим работы с приоритетами* (preemptive multitasking)). С точки зрения пользователя, задачи A, B и C выполняются одновременно, как будто они фактически выполняются одновременно на нескольких ядрах.

Таким образом, однопоточный характер движка JavaScript мог бы показаться ограничивающим фактором, но фактически это освобождает нас от заботы о некоторых очень сложных проблемах, присущих многопоточному программированию. Эта свобода имеет цену: она означает, что для написания корректно выполняющегося программного обеспечения нужно мыслить асинхронно, и не только о пользовательском вводе. Поначалу так мыслить может быть трудно, особенно если вы переходите с языка программирования, в котором выполнение обычно происходит синхронно.

В JavaScript с самого начала был заложен механизм для асинхронного выполнения программ. Однако по мере роста популярности JavaScript (и изощренности создаваемого на нем программного обеспечения) в него были добавлены новые конструкции для асинхронного программирования. Фактически мы можем считать, что JavaScript имеет три разные фазы асинхронной поддержки: фазу обратного вызова, фазу обязательства и фазу генератора. Если бы это был вопрос только генераторов, ставших лучше, чем прежде, мы объяснили бы, как они работают, и двинулись бы дальше.

Но не тут-то было. Генераторы сами по себе не обеспечивают никакой асинхронной поддержки: для обеспечения асинхронного поведения они полагаются либо на обязательства (в русскоязычной документации MDN они названы обещаниями), либо на специальный тип функций обратного вызова. Аналогично обязательства сами по себе столь же полезны, но они полагаются на обратные вызовы (а обратные вызовы сами по себе очень пригодятся для таких вещей, как обработка событий).

Кроме пользовательского ввода, вы будете использовать асинхронные методики в следующих трех областях.

- Сетевые запросы (например, вызов Ajax).
- Операции с файловой системой (чтение, запись в файлы и т.д.).
- Преднамеренно отсроченные функциональные возможности (например, оповещение).

Аналогия

Аналогия, которую мне нравится использовать для функций обратных вызовов и обязательств, — это попытка заполучить столик в занятом ресторане, когда вы заранее не зарезервировали себе место. Таким образом, вы не должны ждать своей очереди (в некоторых ресторанах могут взять номер вашего мобильного телефона и перезвонить, когда столик освободится). Это похоже на функцию обратного вызова: вы предоставили менеджеру ресторана нечто, что позволит ему сообщить, когда ваш столик освободится. Ресторан занят своими делами, и вы можете заниматься своими; никто никого не ждет. В другом ресторане вам могут предоставить пейджер, который сработает, когда столик будет свободен. Это больше похоже на обязательство: нечто, что менеджер ресторана дает вам, и что сообщит, когда столик будет свободен.

Не забывайте эти аналогии, когда мы будем рассматривать обратные вызовы и обязательства, особенно если вы новичок в асинхронном программировании.

Обратные вызовы

Обратный вызов (callback) — самый старый асинхронный механизм в JavaScript, и мы уже встречали его при обработке пользовательского ввода и запуска программ через заданные интервалы времени. Обратный вызов — это просто функция, которую вы пишете и которая будет вызвана в некий момент времени в будущем. В самой функции нет ничего особенного: это обычная функция JavaScript. Как правило, вы предоставляете функции обратного вызова другим функциям или устанавливаете их как свойства объектов (или, хотя и редко, сохраняете их в массиве). Обратные вызовы (часто, но не всегда) являются анонимными функциями.

Давайте начнем с простого примера использования встроенной функции setTimeout, которая задерживает запуск программы на указанное количество миллисекунд.

```
console.log("До таймаута: " + new Date());
function f() {
    console.log("После таймаута: " + new Date());
}
setTimeout(f, 60*1000);             // одна минута
console.log("Это произошло после вызова setTimeout!");
console.log("И это тоже!");
```

Если запустить данный пример на консоли (если только вы не очень медленно его вводите), можно увидеть нечто такое.

```
До таймаута: Sun Aug 02 2015 17:11:32 GMT-0700 (Pacific Daylight Time)
Это произошло после вызова setTimeout!
И это тоже!
После таймаута: Sun Aug 02 2015 17:12:32 GMT-0700 (Pacific Daylight Time)
```

Причина затруднений у новичков кроется в разрыве между линейной природой кода, который мы *пишем*, и фактическим *выполнением* этого кода. Некоторые из нас хотят (или ожидают), чтобы компьютер выполнял код точно в том порядке, в котором он был написан. Другими словами, мы хотели бы увидеть следующее.

```
До таймаута: Sun Aug 02 2015 17:11:32 GMT-0700 (Pacific Daylight Time)
После таймаута: Sun Aug 02 2015 17:12:32 GMT-0700 (Pacific Daylight Time)
Это произошло после вызова setTimeout!
И это тоже!
```

Хотеть не вредно... но от этого код не стал бы асинхронным! Основной момент асинхронного выполнения состоит в том, что оно не должно ничего *блокировать*. Поскольку JavaScript имеет однопоточный характер, то если бы мы указали движку ждать в течение 60 секунд, а затем запустить некоторый код, и сделали бы это синхронно, то в этот момент ничего бы не работало. Ваша программа просто "зависла" бы на это время: она перестала бы реагировать на пользовательский ввод, не обновляла бы экран и т.д. У всех нас были подобные случаи, к сожалению. Асинхронная техника помогает предотвратить данный вид блокировки.

В нашем примере для ясности мы использовали именованную функцию при передаче в setTimeout. Если нет серьезной причины использовать именованную функцию, обычно используют анонимную функцию.

```
setTimeout(function() {
    console.log("После таймаута: " + new Date());
}, 60*1000);
```

Функция setTimeout немного проблематична, поскольку числовой параметр интервала времени является последним аргументом. При использовании анонимных

функций, особенно если они длинны, его можно легко потерять, или он будет выглядеть, как часть функции. Это достаточно распространенное явление, поэтому вы должны привыкнуть к использованию функции `setTimeout` (и ее компаньонки `setInterval`) с анонимными функциями. Только не забывайте, что в последней строке содержится параметр задержки!

Функции `setInterval` и `clearInterval`

Кроме функции `setTimeout`, которая запускает свою функцию один раз и останавливается, есть функция `setInterval`, которая запускает функцию обратного вызова через определенный интервал времени бесконечно или пока вы не вызовете функцию `clearInterval`. Вот пример, в котором код запускается каждые 5 секунд в течение одной минуты, или 10 раз, если это произойдет раньше.

```
const start = new Date();
let i=0;
const intervalId = setInterval(function() {
    let now = new Date();
    if(now.getMinutes() !== start.getMinutes() || ++i>10)
        return clearInterval(intervalId);
    console.log('${i}: ${now}');
}, 5*1000);
```

Здесь мы видим, что `setInterval` возвращает идентификатор, который впоследствии можно использовать для отмены режима интервального запуска кода. Есть соответствующая функция `clearTimeout`, которая работает так же и позволяет сбросить интервал времени и предотвратить запуск кода.

Функции `setTimeout`, `setInterval` и `clearInterval` определены в глобальном объекте (`window` в браузере и `global` в Node).

Область видимости и асинхронное выполнение

Распространенным источником беспорядка (и ошибок) в асинхронном выполнении является то, как области видимости и замкнутые выражения влияют на асинхронное выполнение. Каждый раз, вызывая функцию, вы создаете замкнутое выражение: все переменные, которые создаются в функции (включая аргументы), существуют, пока что-то может к ним обращаться.

Мы видели этот пример прежде, но его имеет смысл повторить для важного урока, который мы можем из него извлечь. Рассмотрим пример функции `countdown`. Наша цель — создать 5-секундный обратный отсчет.

```
function countdown() {
    let i;       // заметьте, что мы объявляем let за пределами цикла for
```

```
    console.log("Обратный отсчет:");
    for(i=5; i>=0; i--) {
        setTimeout(function() {
            console.log(i===0 ? "Старт!" : i);
        }, (5-i)*1000);
    }
}
countdown();
```

Давайте сначала пройдем этот пример мысленно. Вы, вероятно, помните, что здесь что-то не так. Все выглядит так, как будто мы выполняем обратный отсчет от 5 до 0. Вместо этого получаем шесть раз по `-1` и без вывода строки `"Старт!"`. Мы уже видели это, когда использовали `var`; на сей раз мы используем `let`, но в объявлении за пределами цикла `for`, поэтому возникает та же проблема: цикл `for` быстро выполняется полностью, оставляя `i` со значением `-1`, и только *затем* запускает на выполнение функцию обратного вызова. Проблема в том, что, когда она выполняется, `i` уже имеет значение `-1`.

Важный урок здесь заключается в способе, которым область видимости и асинхронное выполнение влияют друг на друга. Вызывая `countdown`, мы создаем замкнутое выражение, которое содержит переменную `i`. Все (анонимные) обратные вызовы, которые мы создаем в цикле `for`, имеют доступ к той же переменной `i`.

Суть этого примера в том, что в цикле `for` мы видим `i`, используемую двумя разными способами. Когда мы используем ее для вычисления периода (`(5-i)*1000`), все работает как ожидалось: первый период — `0`, второй период — `1000`, третий период — `2000` и т.д. Это потому, что вычисление происходит синхронно. Фактически вызов функции `setTimeout` также синхронен. В ней выполняются некие вычисления, позволяющие точно определить момент запуска функции обратного вызова. Асинхронная часть — это функция, которая передается функции `setTimeout`, и именно здесь кроется проблема.

Напомню, что мы можем решить эту проблему, используя немедленно вызываемое функциональное выражением (IIFE), или еще проще, переместив объявление `i` в объявление цикла `for`.

```
function countdown() {
    console.log("Обратный отсчет:");
    for(let i=5; i>=0; i--) {   // теперь i имеет область видимости блока
        setTimeout(function() {
            console.log(i===0 ? "Старт!" : i);
        }, (5-i)*1000);
    }
}
countdown();
```

Урок здесь состоит в том, что нужно помнить области видимости, в которых объявляются ваши функции обратного вызова: у них будет доступ ко всему в этих областях видимости (замкнутое выражение). Именно из-за этого значение переменной может быть отличным от ожидаемого, когда функция обратного вызова выполняется

фактически. Этот принцип применим ко всем асинхронным методикам, а не только к обратным вызовам.

Передача ошибок функциям обратного вызова

В некий момент роста популярности среды Node было принято соглашение *об использовании первого аргумента в функции обратного вызова* для передачи ей ошибок (error-first callback). Поскольку, как мы вскоре увидим, механизм обратных вызовов затрудняет обработку исключений, нужен стандартный способ сообщения о проблеме, возникшей в момент запуска функции обратного вызова. Соглашение подразумевает использование первого аргумента функции обратного вызова для доступа к объекту ошибки. Если значение этого аргумента `null` или `undefined`, никакой ошибки не было.

Всякий раз, когда вы имеете дело с функцией обратного вызова с первым аргументом для передачи ошибки, первое, что нужно сделать, — проверить его на наличие ошибки и выполнить соответствующее действие. Рассмотрим попытку чтения содержимого файла в Node, приводящую к ошибке, для обработки которой используется соглашение о передаче ошибок функциям обратного вызова.

```
const fs = require('fs');

const fname = 'may_or_may_not_exist.txt';
fs.readFile(fname, function(err, data) {
    if(err) return console.error('Ошибка при чтении файла ${fname}: ${err.mes-
sage}');
    console.log('${fname} содержит: ${data}');
});
```

Первое, что мы делаем в функции обратного вызова, — проверяем переменную `err` на истинность. Если это так, значит, при чтении файла возникла проблема, и мы выводим на консоль сообщение об этом, а затем *немедленно выходим* (метод `console.error` не возвращает никакого смыслового значения, и мы не используем его ни коим образом, поэтому мы можем все объединить в одном операторе). При использовании описанного выше механизма наиболее часто допускаемой ошибкой, вероятно, является случай, когда программист, после проверки, а возможно, и вывода сообщения об ошибке, забывает о том, что нужно немедленно выйти из функции. Если этого не сделать и позволить функции продолжить выполняться, она будет считать, что в момент ее вызова не возникло никаких проблем, соответственно результат работы функции обратного вызова будет непредсказуем. Разумеется, возможен случай, когда в функции обратного вызова предусмотрена специальная ветка, которая должна выполняться в случае ошибки. Тогда после анализа переданного аргумента на предмет ошибки и ее фиксации можно продолжить выполнение функции.

Соглашение о передаче ошибок в функцию обратного вызова стало де-факто стандартом при разработке программ для Node (когда обязательства не используются),

и если вы пишете интерфейс, подразумевающий получение функции обратного вызова, я настоятельно рекомендую придерживаться этого соглашения.

Проклятье обратных вызовов

Хотя обратные вызовы позволяют управлять асинхронным выполнением, у них есть практический недостаток: они с трудом справляются с ситуацией, когда необходимо ждать завершения нескольких процессов перед продолжением. Вообразите случай, когда вы пишете приложение для Node, которое должно получить содержимое трех разных файлов, а затем выждать 60 секунд, прежде чем объединить содержимое этих файлов и записать в четвертый файл.

```
const fs = require('fs');

fs.readFile('a.txt', function(err, dataA) {
    if(err) console.error(err);
    fs.readFile('b.txt', function(err, dataB) {
        if(err) console.error(err);
        fs.readFile('c.txt', function(err, dataC) {
            if(err) console.error(err);
            setTimeout(function() {
                fs.writeFile('d.txt', dataA+dataB+dataC, function(err) {
                    if(err) console.error(err);
                });
            }, 60*1000);
        });
    });
});
```

Программисты называют это *проклятьем обратных вызовов* (callback hell). Его признаком является треугольная форма блоков кода, образованных вложенными фигурными скобками. Проблема обработки ошибок здесь стоит еще острее. Все, что мы делаем в этом примере, — это регистрируем ошибки, но если бы мы попытались сгенерировать исключение, то были бы неприятно удивлены. Рассмотрим следующий упрощенный пример.

```
const fs = require('fs');
function readSketchyFile() {
    try {
        fs.readFile('does_not_exist.txt', function(err, data) {
            if(err) throw err;
        });
    } catch(err) {
        console.log('Внимание: возникли небольшие проблемы, продолжаем
выполнение программы');
    }
}
readSketchyFile();
```

На первый взгляд, все кажется достаточно резонным и не вызывает неодобрения у программистов, использующих обработку исключений. Но только это не будет работать. Давайте опробуем эту программу. Она завершится аварийно, даже при том что мы проявили столько заботы, чтобы гарантировать отсутствие проблем из-за этой почти ожидаемой ошибки. Дело в том, что блоки `try...catch` работают только в пределах одной и той же функции. Блок `try...catch` находится в `readSketchyFile`, а ошибка возникает в анонимной функции, которую `fs.readFile` вызывает как функцию обратного вызова.

Кроме того, нет никакой гарантии, что функция обратного вызова не будет вызвана несколько раз (или вообще ни разу!). Если при написании программы вы предполагаете, что она должна вызываться только один раз, в самом языке не предусмотрено никаких средств контроля за этим процессом.

Эта проблема вполне преодолима, но с распространением асинхронного кода она делает написание удобного в сопровождении и безошибочного кода весьма затруднительным. Вот здесь и пригодятся обязательства.

Обязательства

Обязательства[1] (promise) пытаются устранить некоторые из недостатков функций обратного вызова. Используя обязательства (хотя это и не всегда просто), можно получить более безопасный и "простой в сопровождении" код.

Обязательства не заменяют функций обратного вызова; фактически с обязательствами вы все еще должны использовать обратные вызовы. Что на самом деле делают обязательства, так это гарантируют единообразный и предсказуемый способ обработки обратных вызовов, устраняя некоторые из нежелательных неожиданностей и трудно обнаруживаемых ошибок, которые можно получить, используя только функции обратного вызова.

Основная идея обязательств проста: когда вы вызываете асинхронную функцию на базе обязательства, она возвращает экземпляр объекта `Promise`. С этим обязательством могут случиться только две вещи: оно может быть *выполнено* (fulfilled) (в случае успеха) или *отклонено* (rejected) (в случае неудачи). Вам гарантируется, что произойдет *только одно* из этих событий (обязательство не может сначала быть выполнено, а затем отклонено) и будет получен только *один* результат. Если

[1] В русскоязычной документации MDN термин promise переведен как обещание. Однако по смыслу, который вложили в этот термин разработчики языка, — это именно обязательство! Имеется в виду, что создавая обязательство и возвращая его в исходный код, интерпретатор JavaScript обязуется в дальнейшем при наступлении нужного события либо выполнить его, либо отклонить. Причем только однократно! Русскоязычный же термин обещание несет на себе некий оттенок необязательности, который плохо сочетается со строгими рамками работы языка программирования. Поэтому в дальнейшем в книге мы будем использовать термин обязательство. К тому же это одно из значений английского слова promise. Переводчики документации MDN почему то решили взять его самое первое значение. — *Примеч. ред.*

обязательство будет выполнено, то оно будет выполнено только однократно; если оно будет отклонено, то тоже только однажды. Как только обязательство было либо выполнено, либо отклонено, оно считается *завершенным* (settled).

Еще одно преимущество обязательств по сравнению с обратными вызовами заключается в том, что, обязательства — это обычные объекты, которые легко можно передать в другие функции. Если вы хотите начать асинхронный процесс, но предпочли бы, чтобы результаты обрабатывал кто-то другой, можете просто передать обязательство ему (это походило бы на передачу пейджера для резервирования столика вашему другу — ресторан не будет возражать, ему безразлично, кто именно получит столик, если количество посетителей за ним останется прежним).

Создание обязательств

Создание обязательств является простым делом: вам нужно создать новый экземпляр объекта `Promise` с функцией, которой передаются две функции обратного вызова `resolve` (выполнено) и `reject` (отклонено) (я предупреждал вас, что обязательства не спасают от обратных вызовов!). Давайте возьмем нашу функцию `countdown`, параметризируем ее (чтобы не зацикливаться только на 5-секундном обратным отсчете) и сделаем так, чтобы она возвратила обязательство, когда обратный отсчет начнется.

```
function countdown(seconds) {
    return new Promise(function(resolve, reject) {
        for(let i=seconds; i>=0; i--) {
            setTimeout(function() {
                if(i>0) console.log(i + '...');
                else resolve(console.log("Старт!"));
            }, (seconds-i)*1000);
        }
    });
}
```

Прямо сейчас эта функция не очень гибка. Дело в том, что нам не всегда требуется вербальный вывод, более того, нам не всегда требуется выводить что-то на консоль. Такой подход совершенно не годится, если мы планируем выводить информацию на веб-странице, модифицируя соответствующий элемент DOM с помощью нашей функции обратного отсчета. Но это только начало... и демонстрация создания обязательств. Обратите внимание, что `resolve` (как и `reject`) является функцией. Вы могли бы подумать "Ха-ха! Я могу вызвать `resolve` несколько раз и нарушить... обязательство обязательств". Вы действительно можете вызывать функции `resolve` или `reject` многократно или даже попеременно... но будет учитываться только их первый вызов. Обязательство гарантирует, что кто бы его ни использовал, он получит

только одно событие — выполнение или отклонение (в настоящее время в нашей функции нет ветки для реализации отклонения).

Использование обязательств

Давайте посмотрим, как мы можем использовать свою функцию `countdown`. Мы могли бы только вызвать ее и проигнорировать обязательство вообще: `countdown(5)`. В результате мы также получим свой обратный отсчет и можем не возиться с обязательствами вообще. Но что если необходимо воспользоваться преимуществами обязательств? Вот как мы используем возвращаемое обязательство.

```
countdown(5).then(
    function() {
        console.log("Обратный отчет завершен");
    },
    function(err) {
        console.log("Ошибка при обратном отсчете: " + err.message);
    }
);
```

В этом примере мы не потрудились присвоить возвращенное обязательство переменной; мы просто вызвали его (метод `then`) обработчик непосредственно. Этому обработчику передается две функции обратного вызова: первая вызывается при выполнении обязательства (т.е. при нормальном завершении), а вторая — при его отклонении (т.е. при возникновении ошибки). Причем вызвана будет только одна из этих функций. Обязательства поддерживают также обработчик `catch`. Таким образом, вы можете разделить эти два обработчика (мы также сохранили обязательство в переменной, чтобы продемонстрировать это).

```
const p = countdown(5);
p.then(function() {
    console.log("Обратный отчет завершен");
});
p.catch(function(err) {
    console.log("Ошибка при обратном отсчете: " + err.message);
});
```

Давайте изменим нашу функцию `countdown` так, чтобы создать условие для возникновения ошибки. Предположим, что мы суеверны и считаем ошибкой случай, если при счете встретится число 13.

```
function countdown(seconds) {
    return new Promise(function(resolve, reject) {
        for(let i=seconds; i>=0; i--) {
            setTimeout(function() {
                if(i===13) return reject(new Error("Принципиально это не
считаем!"));
```

```
        }
    });
}
```

Давайте поэкспериментируем с этим примером. Обратите внимание на его интересное поведение. Вполне очевидно, что вы можете считать в обратном порядке от любого числа, меньшего чем 13, и поведение будет обычным. Обратный отсчет от 13 или больше должен завершиться неудачей, когда дойдет до 13. Однако... вывод на консоль будет продолжаться! Вызов `reject` (или `resolve`) не останавливает нашу функцию; он только управляет состоянием обязательства.

Конечно, наша функция `countdown` нуждается в некоторых усовершенствованиях. Обычно нам не нужно, чтобы функция продолжала работать после завершения обязательства (успешного или нет), а наша продолжает. Мы также уже упомянули, что вывод на консоль не очень гибок. Он действительно не дает нам желаемого контроля.

Обязательства дают нам чрезвычайно четкий и безопасный механизм для асинхронных задач, которые либо выполняются, либо отклоняются, но они (пока что!) не предоставляют никакого способа сообщения о ходе выполнения *самого процесса*. Таким образом, обязательство либо выполняется, либо нет. Мы никогда не узнаем, что оно выполнено "только на 50%". В некоторых библиотеках обязательств[2] реализована очень полезная возможность сообщать о ходе выполнения процесса, и вполне возможно, что в будущем эти функциональные возможности появятся в JavaScript, но пока что мы должны уметь обходиться без этого, что плавно подводит нас к следующему разделу.

События

События (event) — это еще одна старая идея, которая получила продолжение в JavaScript. Концепция событий проста: эмиттер (источник) события передает сообщение о событии, а любой, кто желает услышать (или "подписаться") об этом событии, может сделать это. Как подписаться на событие? Используя функцию обратного вызова, конечно! Создать собственную систему событий очень просто, но Node обеспечивает их встроенную поддержку. Если вы работаете в браузере, jQuery также предоставляет механизм событий. Чтобы улучшить функцию `countdown`, мы будем использовать класс `EventEmitter` от Node. Хотя вполне можно использовать `EventEmitter` с такой функцией, как `countdown`, он предназначен для использоваться с классом. Таким образом, мы превратим свою функцию `countdown` в класс `Countdown`.

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {
    constructor(seconds, superstitious) {
```

[2] Например, Q.

```
        super();
        this.seconds = seconds;
        this.superstitious = !!superstitious;
    }
    go() {
        const countdown = this;
        return new Promise(function(resolve, reject) {
            for(let i=countdown.seconds; i>=0; i--) {
                setTimeout(function() {
                   if(countdown.superstitious && i===13)
                     return reject(new Error("Принципиально это не считаем!"));
                    countdown.emit('tick', i);
                    if(i===0) resolve();
                }, (countdown.seconds-i)*1000);
            }
        });
    }
}
```

Класс `Countdown` наследует класс `EventEmitter`, который позволяет ему генерировать события. Метод `go` — это то, что фактически запускает обратный отсчет и возвращает обязательство. Обратите внимание, что первое, что мы делаем в методе `go`, — это присваиваем значение `this` константе `countdown`. Дело в том, что для получения длины обратного отсчета необходимо использовать значение `this` вне зависимости от того, суеверен ли обратный отсчет *в обратных вызовах*. Помните, что `this` — это специальная переменная, и у нее не будет того же значения в функции обратного вызова. Таким образом, следует сохранить текущее значение `this`, чтобы можно было использовать его в обязательствах.

Магия происходит при вызове `countdown.emit('tick', i)`. Любой, кто хочет услышать о событии `tick` (мы могли бы называть его как угодно по своему усмотрению; на мой взгляд, "tick" не хуже других), может сделать это. Давайте посмотрим, как можно использовать этот новый, улучшенный обратный отсчет.

```
const c = new Countdown(5);

c.on('tick', function(i) {
    if(i>0) console.log(i + '...');
});

c.go()
    .then(function() {
        console.log('Старт!');
    })
    .catch(function(err) {
        console.error(err.message);
})
```

Метод on класса EventEmitter как раз и позволяет прослушивать сообщения о событии. В этом примере мы предоставляем обратный вызов для каждого события tick. Если tick не 0, мы выводим его. Затем происходит вызов метода go, который запускает обратный отсчет. Когда обратный отсчет заканчивается, мы выводим строку "Старт!". Конечно, мы могли бы поместить вывод "Старт!" в обработчик события tick, но так мы подчеркнули различие между событиями и обязательствами.

Результат, определенно, более подробен, чем наша первоначальная функция countdown, и мы получили намного больше функциональных возможностей. Теперь у нас есть полный контроль над регистрацией событий при обратном отсчете и обязательство, выполняемое при завершении обратного отсчета.

Мы все еще имеем в запасе одну задачу — мы не решили проблему суеверного экземпляра Countdown, продолжающего обратный отсчет после 13, даже при том что обязательство было отклонено.

```
const c = new Countdown(15, true)
    .on('tick', function(i) {  // заметьте, мы можем "сцепить" вызов 'on'
        if(i>0) console.log(i + '...');
    });

c.go()
    .then(function() {
        console.log('Старт!');
    })
    .catch(function(err) {
        console.error(err.message);
})
```

Мы получим сообщение о событии tick даже если будет достигнуто значение 0 (хотя и не выводим информации о нем). Решение этой проблемы немного затруднительно, поскольку мы уже создали все нужные нам интервалы времени. Конечно, здесь мы могли бы просто "смошенничать" и немедленно прервать работу, если суеверный таймер создается для отсчета 13 или более секунд, но это противоречило бы задаче упражнения! Для решения данной проблемы, поскольку мы обнаружили, что не можем без этого продолжать, мы должны будем очистить все ожидающие обработки интервалы времени.

```
const EventEmitter = require('events').EventEmitter;

class Countdown extends EventEmitter {
    constructor(seconds, superstitious) {
        super();
        this.seconds = seconds;
        this.superstitious = !!superstitious;
    }
    go() {
```

```
        const countdown = this;
        const timeoutIds = [];
        return new Promise(function(resolve, reject) {
            for(let i=countdown.seconds; i>=0; i--) {
                timeoutIds.push(setTimeout(function() {
                    if(countdown.superstitious && i===13) {
                        // очистить все ожидающие обработки периоды
                        timeoutIds.forEach(clearTimeout);
                        return reject(new Error("Принципиально это не
считаем!"));
                    }
                    countdown.emit('tick', i);
                    if(i===0) resolve();
                }, (countdown.seconds-i)*1000));
            }
        });
    }
}
```

Сцепление обязательств

Одно из преимуществ обязательств в том, что они могут быть *сцеплены* (chained), т.е. когда одно обязательство выполняется, вы можете немедленно вызвать другую функцию, которая возвращает обязательство... и т.д. Давайте создадим функцию `launch`, которую мы можем сцепить с обратным отсчетом.

```
function launch() {
    return new Promise(function(resolve, reject) {
        console.log("Поехали!");
        setTimeout(function() {
            resolve("На орбите!");
        }, 2*1000);                // действительно очень быстрая ракета
    });
}
```

Сцепить эту функцию с обратным отсчетом довольно просто.

```
const c = new Countdown(5)
    .on('tick', i => console.log(i + '...'));

c.go()
    .then(launch)
    .then(function(msg) {
        console.log(msg);
    })
    .catch(function(err) {
        console.error("Хьюстон, у нас проблемы....");
    })
```

Одно из преимуществ сцепления обязательств в том, что вы не обязаны обрабатывать ошибки на каждом этапе; если где-нибудь в цепочке произойдет ошибка, то цепочка остановится и управление перейдет к обработчику `catch`. Давайте заменим обратный отсчет 15-секундным суеверным обратным отсчетом; вы обнаружите, что функция `launch` никогда не будет вызвана.

Предотвращение незавершенных обязательств

Обязательства могут упростить ваш асинхронный код и защитить вас от проблем функций обратных вызовов, вызываемых несколько раз, но они не защищают вас от проблемы обязательств, которые никогда не завершаются (т.е. когда вы забываете вызвать `resolve` или `reject`). Ошибки этого вида может быть очень трудно обнаружить, поскольку никакой ошибки нет... а в сложной системе незавершенное обязательство может быть просто потеряно.

Один из способов предотвращения этого заключается в определении периода для обязательств; если обязательство не завершено за некий разумный период времени, оно автоматически отклоняется. Вполне очевидно, что это вам решать, каков "разумный период времени". Если у вас достаточно сложный алгоритм, выполнение которого предположительно займет 10 минут, не устанавливайте 1-секундный период.

Давайте вставим в нашу функцию `launch` искусственный отказ. Скажем, наша ракета *очень* экспериментальная, и она отказывает приблизительно в половине случаев.

```
function launch() {
    return new Promise(function(resolve, reject) {
        if(Math.random() < 0.5) return; // отказ ракеты
        console.log("Поехали!");
        setTimeout(function() {
            resolve("На орбите!");
        }, 2*1000);              // действительно очень быстрая ракета
    });
}
```

Способ отказа в данном примере не очень достойный: мы не вызываем функцию `reject` и даже ничего не выводим на консоль. Мы просто тихо выходим в половине случаев. Если запустить этот пример несколько раз, то можно увидеть, что иногда он срабатывает, а иногда нет... безо всякого сообщения об ошибке, что явно нежелательно.

Мы можем написать функцию, которая задает обязательству период.

```
function addTimeout(fn, timeout) {
    if(timeout === undefined) timeout = 1000; // стандартный период
    return function(...args) {
        return new Promise(function(resolve, reject) {
            const tid = setTimeout(reject, timeout,
                new Error("Истек период обязательства"));
            fn(...args)
```

```
    }
}
```

Если вы уже говорите "Ого... функция, которая возвращает функцию, которая возвращает обязательство, которое вызывает функцию, которая возвращает обязательство... У меня уже голова кружится!", я могу вас понять: добавление периода в возвращающую обязательство функцию не является тривиальной задачей и требует напряжения всех извилин. Полное понимание этой функции является упражнением для "продвинутого" читателя. Однако использовать эту функцию весьма просто: мы можем добавить период к любой функции, которая возвращает обязательство. Скажем, наша самая медленная ракета достигает орбиты через 10 секунд (разве не прекрасны ракетные технологии будущего?). Таким образом, мы устанавливаем период на 11 секунд.

```
c.go()
    .then(addTimeout(launch, 4*1000))
    .then(function(msg) {
        console.log(msg);
    })
    .catch(function(err) {
        console.error("Хьюстон, у нас проблемы: " + err.message);
});
```

Теперь наша цепь обязательств всегда будет завершаться, даже когда функция `launch` ведет себя плохо.

Генераторы

Как уже обсуждалось в главе 12, генераторы обеспечивают двухстороннюю связь между функцией и ее вызывающей стороной. Генераторы синхронны по своей природе, но, будучи объединены с обязательствами, обеспечивают мощную технологию для управления асинхронным кодом в JavaScript.

Давайте припомним главную сложность асинхронного кода: его труднее писать, чем синхронный код. Когда мы решаем задачу, наш ум стремится свести ее к синхронному виду: этап 1, этап 2, этап 3 и т.д. Однако при этом подходе могут быть проблемы производительности, которых нет при асинхронном подходе. Разве не было

бы хорошо иметь преимущества производительности асинхронных технологий без дополнительных концептуальных трудностей? Вот где могут пригодиться генераторы.

Рассмотрим использованный ранее пример "проклятья обратных вызовов": чтение трех файлов, задержка на одну минуту и последующая запись содержимого первых трех файлов одного за другим в четвертый файл. Наш человеческий разум *хотел бы* написать это в виде примерно такого псевдокода.

```
dataA = читаем содержимое файла 'a.txt'
dataB = читаем содержимое файла 'b.txt'
dataC = читаем содержимое файла 'c.txt'
Ждем 60 секунд
Записываем dataA + dataB + dataC в файл 'd.txt'
```

Генераторы позволяют нам писать код, который выглядит очень похоже на этот... однако необходимые функциональные возможности не появятся как из коробки: сначала придется проделать небольшую работу.

Первое, что необходимо, — это способ превратить функцию обратного вызова с первым аргументом для передачи ошибки Node в обязательство. Мы инкапсулируем это в функцию `nfcall` (Node function call — вызов функции Node).

```
function nfcall(f, ...args) {
    return new Promise(function(resolve, reject) {
        f.call(null, ...args, function(err, ...args) {
            if(err) return reject(err);
            resolve(args.length<2 ? args[0] : args);
        });
    });
}
```

Эта функция названа в честь метода `nfcall` из *библиотеки обязательств Q* (и реализована на его основе). Если вам необходимы эти функциональные возможности, лучше всего воспользуйтесь библиотекой Q. Она включает не только этот метод, но и много других полезных методов, также связанных с обязательствами. Я же представляю реализацию `nfcall` для демонстрации того, что никакого "волшебства" здесь нет.

Теперь мы можем преобразовать любой метод, написанный в стиле Node, так, чтобы он получал обратный вызов для обязательства. Нам также понадобится функция `setTimeout`, которая получает обратный вызов... но поскольку она появилась задолго до Node, она не соответствует соглашению о передаче ошибок функциям обратного вызова. Поэтому необходимо создать новую функцию `ptimeout` (promise timeout — период обязательства).

```
function ptimeout(delay) {
    return new Promise(function(resolve, reject) {
        setTimeout(resolve, delay);
    });
}
```

Следующее, что нам понадобится, — это *пускатель генератора* (generator runner). Не забывайте, что генераторы не являются по своей природе асинхронными. Но поскольку они позволяют функциям общаться с вызывающей стороной, мы можем создать функцию, которая будет управлять этой связью и знать, как обрабатывать асинхронные вызовы. Поэтому создадим функцию `grun` (generator run — пуск генератора).

```
function grun(g) {
    const it = g();
     (function iterate(val) {
        const x = it.next(val);
        if(!x.done) {
            if(x.value instanceof Promise) {
                x.value.then(iterate).catch(err => it.throw(err));
            } else {
                setTimeout(iterate, 0, x.value);
            }
        }
     })();
}
```

Данная функция `grun` основана на функции `runGenerator`, представленной Кайлом Симпсоном (Kyle Simpson) в его превосходной *серии статей о генераторах*. Я настоятельно рекомендую прочитать эти статьи как дополнение к данному тексту.

Это очень скромный рекурсивный пускатель генератора. Вы передаете ему функцию генератора, и он запускает его. Как уже было сказано в главе 6, генераторы, которые вызывают оператор `yield`, будут делать паузу, пока не будет вызван метод `next` его итератора. Данная функция делает это рекурсивно. Если итератор возвращает обязательство, она ожидает выполнения обязательства перед возобновлением итератора. С другой стороны, если итератор возвращает простое значение, она немедленно возобновляет итерацию. Вы можете задаться вопросом "Почему происходит вызов `setTimeout` вместо обычного непосредственного вызова `iterate`?" Причина в том, что мы получим более эффективный код при отказе от синхронной рекурсии (асинхронная рекурсия позволяет движку JavaScript освобождать ресурсы куда быстрее).

Вы можете подумать "Как много суеты!" и "Это называется *упрощает* жизнь?" Однако сложная часть завершена. Функция `nfcall` позволяет увязать прошлое (функции

обратного вызова с первым аргументом для передачи ошибки в стиле Node) с настоящим (обязательства), а функция `grun` обеспечивает доступ к будущему уже сегодня (в спецификации ES7 ожидается ключевое слово `await`, которое будет, по существу, функцией `grun` с еще более естественным синтаксисом). Теперь, когда самая трудная часть закончена, давайте посмотрим, как все это упрощает нашу жизнь.

Помните наш псевдокод "разве не было бы хорошо" ранее в этой главе? Теперь мы можем его реализовать.

```
function* theFutureIsNow() {
    const dataA = yield nfcall(fs.readFile, 'a.txt');
    const dataB = yield nfcall(fs.readFile, 'b.txt');
    const dataC = yield nfcall(fs.readFile, 'c.txt');
    yield ptimeout(60*1000);
    yield nfcall(fs.writeFile, 'd.txt', dataA+dataB+dataC);
}
```

Выглядит намного лучше, чем проклятье обратных вызовов, не так ли? Это куда аккуратнее, чем одни только обязательства, и напоминает способ, которым мы думаем. Используется это так же просто.

```
grun(theFutureIsNow);
```

Шаг вперед и два назад?

Вы могли бы (и весьма резонно) полагать, что мы зашли в *такие глубокие дебри*, только чтобы понять природу асинхронного выполнения и сделать все проще... а теперь мы вернулись к тому, с чего начали, кроме дополнительных сложностей с генераторами и преобразованием всего в обязательства, а также функции `grun`. И в этом есть некоторая доля правды: в нашей функции `theFutureIsNow` ребенка выплеснули с грязной водой несколько раз. Мы добились достаточно простого в написании и чтении кода. Но мы получили лишь *часть* преимуществ от асинхронного выполнения, но не их все. Здесь вполне резонен вопрос "А не было бы эффективнее читать эти три файла параллельно?" Ответ на этот вопрос зависит от конкретной задачи, реализации вашего движка JavaScript, вашей операционной системы и вашей файловой системы. Но давайте отложим эти сложности на мгновение и уясним, что *не имеет значения*, в каком порядке мы будем читать эти три файла, и что выигрыш в эффективности зависит от способности выполнения операций чтения файлов параллельно операционной системой. Именно здесь пускатели генераторов могут создать иллюзию ложной простоты: ведь мы написали функцию именно в таком стиле только потому, что этот путь казался нам простым и очевидным.

Проблема (предположим, что это проблема) решается просто. В классе `Promise` есть метод `all`, который будет завершен (resolves), когда будут завершены (resolve) все обязательства в массиве... и выполняет асинхронный код параллельно, если это

возможно. Остается только модифицировать нашу функцию так, чтобы использовать метод `Promise.all`.

```
function* theFutureIsNow() {
    const data = yield Promise.all([
        nfcall(fs.readFile, 'a.txt'),
        nfcall(fs.readFile, 'b.txt'),
        nfcall(fs.readFile, 'c.txt'),
    ]);
    yield ptimeout(60*1000);
    yield nfcall(fs.writeFile, 'd.txt', data[0]+data[1]+data[2]);
}
```

Обязательство, возвращенное методом `Promise.all`, представляет собой массив, содержащий значение состояния выполнения каждого обязательства *в порядке их расположения в массиве*. Даже при том, что файл `c.txt` вполне может быть прочитан прежде, чем файл `a.txt`, элемент `data[0]` будет все еще хранить содержимое `a.txt`, а `data[2]` — содержимое `c.txt`.

Из этого раздела вам уже должно было стать понятно, что в основе всего лежит не метод `Promise.all` (хотя это и весьма удобный инструмент), а то, что следует *учитывать, какие именно части вашей программы могут быть выполнены параллельно, а какие не могут*. В этом примере вполне можно даже запустить интервальный таймер параллельно с чтением файла: все это зависит от задачи, которую вы пытаетесь решить. Если важно, чтобы эти три файла были прочитаны, *затем* прошло 60 секунд, а *затем* результат их объединения записан в другой файл, то мы уже имеем то, что хотим. С другой стороны, нам может понадобиться, чтобы эти три файла были прочитаны *не ранее, чем через 60 секунд*, а результат был записан в четвертый файл — в этом случае нам нужно переместить установку интервала в метод `Promise.all`.

Не пишите собственных пускателей генераторов

Хотя написание собственного пускателя генератора, как это было сделано с `grun`, является хорошим упражнением, в него следовало бы внести много нюансов и усовершенствований. Лучше не изобретать колесо. *Пускатель генератора* `co` полноценен и надежен. Если вы создаете веб-сайты, вам может иметь смысл изучить *Koa*, который предназначен для работы с `co` и позволяет писать веб-обработчики, используя `yield`, как в функции `theFutureIsNow`.

Обработка исключений в пускателях генераторов

Другое важное преимущество пускателей генератора состоит в том, что они допускают возможность обработки исключений с помощью блоков `try/catch`. Помните, что при использовании функций обратного вызова и обязательств обработка

исключений довольно проблематична. Генерирование исключения в функции обратного вызова не может быть обработано за пределами этой функции. У пускателей генератора, поскольку они допускают синхронную семантику при все еще асинхронном выполнении, есть дополнительное преимущество при работе с `try/catch`. Давайте добавим в нашу функцию `theFutureIsNow` несколько обработчиков исключений.

```
function* theFutureIsNow() {
    let data;
    try {
        data = yield Promise.all([
            nfcall(fs.readFile, 'a.txt'),
            nfcall(fs.readFile, 'b.txt'),
            nfcall(fs.readFile, 'c.txt'),
        ]);
    } catch(err) {
        console.error("Ошибка при чтении файлов: " + err.message);
        throw err;
    }
    yield ptimeout(60*1000);
    try {
        yield nfcall(fs.writeFile, 'd.txt', data[0]+data[1]+data[2]);
    } catch(err) {
        console.error("Ошибка при записи файла: " + err.message);
        throw err;
    }
}
```

Я не утверждаю, что обработка исключений с помощью конструкции `try... catch` непременно превосходит обработчики `catch` в обязательствах или функции обратного вызова с первым аргументом для передачи ошибки, но это хорошо понятый механизм для обработки исключений, и если вы предпочитаете синхронную семантику, то можете использовать ее для обработки исключений.

Заключение

Полное понимание сложностей асинхронного программирования (и различных механизмов, выработанных для управления им) критически важно для понимания современной разработки на JavaScript. В этой главе мы изучили следующие вопросы.

- Управление асинхронным выполнением в JavaScript осуществляется функциями обратного вызова.
- Обязательства не заменяют обратные вызовы; на самом деле они требуют функций обратного вызова `then` и `catch`.

- Обязательства устраняют проблему многократного вызова функций обратного вызова.
- Если необходим многократный вызов функций обратного вызова, рассмотрите возможность использования событий (которые можно комбинировать с обязательством).
- Обязательства могут быть сцеплены, что упрощает композицию.
- Обязательства могут быть объединены с пускателями генераторов, чтобы позволить синхронную семантику, не теряя преимуществ асинхронного выполнения.
- При написании функций генератора с синхронной семантикой следует быть внимательными и четко понимать, какие части вашего алгоритма могут выполняться параллельно с использованием метода `Promise.all`.
- Вам не стоит писать собственные пускатели генераторов; используйте `co` или `Koa`.
- Вам не стоит писать собственный код для преобразования обратных вызовов в стиле Node в обязательства; используйте `Q`.
- Обработка исключений работает с синхронной семантикой, как позволяют пускатели генераторов.

Если вы имеете опыт программирования только на языках с синхронной семантикой, изучение способов асинхронного программирования на JavaScript может привести вас в замешательство, как это было со мной. Однако это необходимый навык в современных проектах JavaScript.

Дата и время

В большинстве реальных приложений используются данные о дате и времени. К сожалению, объект JavaScript `Date` (который хранит данные о дате и времени) не является одним из шедевров языка. Из-за ограниченного удобства этого встроенного объекта я буду использовать `Moment.js`, который наследует функциональные возможности объекта `Date`, для реализации наиболее популярных функциональных возможностей.

Интересный исторический факт: объект JavaScript `Date` был первоначально реализован программистом Netscape Кеном Смитом (Ken Smith); он, по существу, перенес в JavaScript реализацию `java.util.Date` из языка Java. Таким образом, утверждение, что язык JavaScript не имеет никакого отношения к Java, не *полностью* соответствует действительности: если вас когда-либо спросят, имеют ли эти языки что-то общее, можете сказать "Очень немногое, кроме объекта `Date` и общего синтаксического предка".

Поскольку регулярно повторять слова "дата и время" утомительно, далее я буду использовать термин "дата". Дата без явного указания времени будет подразумевать 12:00 утра текущего дня.

Даты, часовые пояса, временные метки и эпохи Unix

Давайте посмотрим правде в глаза: наш современный Григорианский календарь — капризный, сверхсложный, нумеруемый начиная с 1, с нечетной кратностью времени и високосными годами. Часовые пояса добавляют еще больше сложности. Однако эта система по большей части универсальна и нам с ней жить.

Давайте начнем с чего-то простого: с секунд. В отличие от сложного деления времени в Григорианском календаре, секунды просты. Дата и время (представленные в секундах) являются одним числом, аккуратно упорядоченным на числовой оси. Поэтому представление даты и времени в секундах идеально подходит для вычислений. Но для коммуникаций между людьми подходит не очень хорошо: "Эй, Байрон, пообедаем в 1437595200?" (1437595200 — это среда, 22 июля 2015 года, 1 час по полудни тихоокеанского времени.) Но если даты представляются секундами, то чему

соответствует дата 0? Это не дата рождения Христа, а просто произвольная дата: 1 января 1970 года, 00:00:00 UTC.

Поскольку вы, вероятно, знаете, что мир делится на *часовые пояса* (TZ), независимо от того, где вы находитесь, в 7 утра будет утро, а в 7 после полудня — вечер. Часовые пояса могут быть сложны, тем более если учитывать летнее время. Я не буду пытаться объяснить в этой книге все нюансы Григорианского календаря или часовых поясов — Википедия превосходно решает эту задачу. Но чтобы помочь вам понять объект JavaScript `Date` (и пользу `Moment.js`), некоторые из основ рассмотреть стоит.

Все часовые пояса определяются как смещения от *Всемирного координированного времени* (сокращенно — UTC), а все подробности ищите в Википедии. Иногда UTC (не совсем корректно) называют Средним временем по Гринвичу (Greenwich Mean Time — GMT). Например, я в настоящее время нахожусь в Орегоне, который находится в Тихоокеанском часовом поясе. Тихоокеанское время на семь или восемь часов отстает от UTC. Что значит "на семь *или* восемь"? Как это возможно? Все зависит от времени года. Летом это летнее время, и смещение — семь часов. Остальную часть года это стандартное время, и смещение — восемь часов. Здесь важно не запомнить часовые пояса, а понять, как представляются смещения. Если я открою терминал JavaScript и введу **`new Date()`**, я увижу следующее.

```
Sat Jul 18 2015 11:07:06 GMT-0700 (Pacific Daylight Time)
```

Обратите внимание, что в этом очень подробном формате часовой пояс определяется как смещение и от UTC (`GMT-0700`), и по имени (`Pacific Daylight Time`).

В JavaScript во всех экземплярах объекта `Date` дата и время хранится в виде одиночного числа — количества миллисекунд (не секунд!), прошедших с Эпохи Unix. Обычно JavaScript преобразует это число в удобочитаемую Григорианскую дату, когда вы запросите это (как только что было показано). Если вы хотите увидеть числовое представление, просто используйте метод `valueOf()`.

```
const d = new Date();
console.log(d);           // форматированная Григорианская дата с TZ
console.log(d.valueOf()); // миллисекунды начиная с Эпохи Unix
```

Создание объектов `Date`

Объект `Date` может быть создан четырьмя способами. Без аргументов (как мы уже видели), возвращается просто объект `Date`, представляющий текущую дату. Мы можем также предоставить строку, которую JavaScript попытается проанализировать, или мы можем задать конкретную (локальную) дату в миллисекундах. Вот примеры.

```
// все дальнейшее интерпретируется с учетом местного времени
new Date();                          // текущая дата
```

```
// заметьте, что в JavaScript месяцы отсчитываются от
// нуля: 0=Jan, 1=Feb и т.д.
new Date(2015, 0);                    // 12:00 A.M., Jan 1, 2015
new Date(2015, 1);                    // 12:00 A.M., Feb 1, 2015
new Date(2015, 1, 14);                // 12:00 A.M., Feb 14, 2015
new Date(2015, 1, 14, 13);            // 3:00 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30);        // 3:30 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5);     // 3:30:05 P.M., Feb 14, 2015
new Date(2015, 1, 14, 13, 30, 5, 500); // 3:30:05.5 P.M., Feb 14, 2015

// создание дат из временных меток Эпохи Unix
new Date(0);                        // 12:00 A.M., Jan 1, 1970 UTC
new Date(1000);                     // 12:00:01 A.M., Jan 1, 1970 UTC
new Date(1463443200000);            // 5:00 P.M., May 16, 2016 UTC

// для получения дат до Эпохи Unix используйте отрицательные значения
new Date(-365*24*60*60*1000);       // 12:00 A.M., Jan 1, 1969 UTC

// анализ строк даты (стандартное время — местное)
new Date('June 14, 1903');          // 12:00 A.M., Jun 14, 1903 local time
new Date('June 14, 1903 GMT-0000'); // 12:00 A.M., Jun 14, 1903 UTC
```

Выполняя эти примеры, обратите внимание на то, что результаты, которые вы получите, всегда будут давать местное время. Если вы используете UTC (привет, Тимбукту, Мадриду и Гринвичу!), то результаты, представленные в UTC, будут отличаться от представленных в этом примере. Это демонстрирует нам один из основных недостатков объекта JavaScript `Date`: нет никакого способа указать, в каком часовом поясе он должен быть. Внутренне он всегда будет хранить объекты в формате UTC и представлять их согласно местному времени (которое определяется настройками вашей операционной системы). С учетом назначения JavaScript как языка сценариев для браузеров это традиционно было "правильно". Если вы работаете с датами, то, вероятно, хотите отображать их в часовом поясе пользователя. Однако в связи с глобальным характером Интернета (и переносом JavaScript на сервер в виде проекта Node) необходима более надежная обработка часовых поясов.

Библиотека `Moment.js`

Хотя эта книга о самом языке JavaScript, а не о библиотеках, манипуляции датой — настолько важная и общая задача, что я решил познакомить вас с известной и весьма надежной библиотекой дат `Moment.js`.

Библиотека `Moment.js` бывает двух разновидностей: с поддержкой часового пояса и без нее. Поскольку версия с поддержкой часового пояса значительно больше (у нее есть информация обо всех часовых поясах в мире), вы можете использовать ее и без такой поддержки. Для простоты все изложенное ниже относится к версии

с поддержкой часовых поясов. Если нужна меньшая версия, ознакомьтесь с информацией о ее возможностях по адресу `http://momentjs.com`.

Разрабатывая веб-ориентированный проект, вы можете подключить библиотеку `Moment.js` от CDN, как показано ниже.

```
<script src="//cdnjs.cloudflare.com/ajax/libs/moment-timezone/0.4.0/ ↵
moment-timezone.min.js"></script>
```

Если вы работаете с Node, то можете установить библиотеку `Moment.js`, используя команду `npm install --save moment-timezone`, а затем подключить ее в свой сценарий с помощью функции `require`.

```
const moment = require('moment-timezone');
```

Библиотека `Moment.js` велика и надежна, она обладает всеми функциональными возможностями, необходимыми для манипулирования датой. Более подробная информация об этой библиотеке содержится в ее документации.

Практический подход к датам в JavaScript

Теперь, завершив рассмотрение основ и обладая библиотекой `Moment.js`, давайте применим немного иной подход к изложению этой информации. Исчерпывающий охват методов, доступных в объекте `Date`, был бы сух и не очень полезен для большинства людей. Кроме того, если эта информация необходима, есть исчерпывающая и хорошо написанная библиотека MDN, содержащая полное описание *объекта* `Date`.

Вместо этого в данной книге будет использован подход, напоминающий поваренную книгу, — мы рассмотрим обработку дат в общем, как необходимо большинству людей, а что именно при этом применять, `Date` или `Moment.js`, будет зависеть от обстоятельств.

Создание дат

Мы уже рассматривали доступные для вас возможности создания объектов `Date` в JavaScript, и они по большей части адекватны. Всякий раз, когда вы создаете дату без явного указания часового пояса, полученная дата будет использовать часовой пояс, зависящий от того, *где* создается дата. В прошлом это сбивало с толку многих новичков: они использовали тот же код даты на сервере в Арлингтоне (штат Виргиния), просматривали его в браузере пользователя, подключившегося в Лос-Анджелесе (штат Калифорния), и с удивлением обнаруживали разницу в три часа.

Создание дат на сервере

Если вы создаете даты на сервере, я рекомендую либо использовать UTC, либо явно указывать часовой пояс. При современном основанном на сетевой среде

(облаке) подходе к разработке приложений один и тот же базовый код может выполняться на серверах во всем мире. Создавая локальные даты, вы напрашиваетесь на неприятности. Если вы в состоянии использовать даты UTC, можете создавать их, используя метод `UTC` объекта `Date`.

```
const d = new Date(Date.UTC(2016, 4, 27)); // May 27, 2016 UTC
```

Метод `Date.UTC` получает все те же варианты аргументов, что и конструктор `Date`, но вместо нового экземпляра объекта `Date` он возвращает числовое значение даты. Затем это число может быть передано в конструктор `Date` для создания экземпляра даты.

Если необходимо создавать даты на сервере находящемся в определенном часовом поясе (и нет желания осуществлять преобразования часового пояса вручную), вы можете использовать `moment.tz` для создания экземпляров `Date` с определенным часовым поясом.

```
// Передача массива в Moment.js использует те же параметры, что и
// конструктор Date JavaScript, включая отсчитываемый от нуля месяц
// (0=Jan, 1=Feb и т.д.). Метод toDate() преобразует назад
// в объект Date JavaScript.
const d = moment.tz([2016, 3, 27, 9, 19], 'America/Los_Angeles').toDate();
```

Создание дат в браузере

Вообще, стандартное поведение JavaScript соответствует браузеру. Браузеру от операционной системы известен часовой пояс, в котором он находится, и пользователи обычно предпочитают для работы местное время. Если вы создаете приложение, которое должно обрабатывать даты в других часовых поясах, то лучше использовать `Moment.js` для обработки, преобразования и представления дат в других часовых поясах.

Передача дат

Все становится куда интересней при передаче даты, когда сервер посылает дату в браузер или наоборот. Сервер и браузер могут находиться в разных часовых поясах, а пользователи хотят видеть даты в их локальном часовом поясе. К счастью, поскольку экземпляры `Date` JavaScript хранят дату как числовое смещение времени в UTC от Эпохи Unix, передавать объекты `Date` обычно безопасно.

Мы говорили о "передаче" весьма неопределенно, но что же именно мы под ней подразумеваем? Самый верный способ гарантировать правильность передачи дат в JavaScript — это использовать спецификацию JSON (JavaScript Object Notation). Фактически эта спецификация не определяет тип данных для дат, что весьма прискорбно, поскольку это предотвращает симметричный анализ JSON.

```
const before = { d: new Date() };
before.d instanceof date                // true
const json = JSON.stringify(before);
const after = JSON.parse(json);
after.d instdanceof date                // false
typeof after.d                          // "string"
```

Таким образом, плохая новость в том, что JSON не может полностью и симметрично обрабатывать даты в JavaScript. Хорошая новость в том, что строковая сериализация, которую использует JavaScript, всегда единообразна, поэтому вы можете "восстановить" дату.

```
after.d = new Date(after.d);
after.d instanceof date                 // true
```

Независимо от того, какой именно часовой пояс первоначально использовался при создании даты, после ее перекодировки в JSON она будет в формате UTC, а когда строка кода JSON будет передана конструктору `Date`, дата будет отображена в локальном часовом поясе.

Другой безопасный способ передачи даты между клиентом и сервером подразумевает просто использование числового значения даты.

```
const before = { d: new Date().valueOf() };
typeof before.d                         // "number"
const json = JSON.stringify(before);
const after = JSON.parse(json);
typeof after.d                          // "number"
const d = new Date(after.d);
```

Хотя JavaScript прекрасно работает с перекодировкой JSON дат в строки, библиотеки JSON для других языков и платформ *нет*. Сериализатор JSON для .NET, в частности, заключает кодированные JSON объекты даты в оболочку их собственного формата. Так, если вы взаимодействуете с JSON из другой системы, потрудитесь разобраться, как она сериализирует даты. Если вы контролируете исходный код, то, возможно, безопаснее будет передавать числовые даты как смещения от Эпохи Unix. Но даже в этом случае следует быть внимательным: библиотеки дат зачастую предоставляют числовое значение в секундах, а не в миллисекундах.

Отображение дат

Форматирование дат при выводе зачастую является одной из самых раздражающих задач для новичков. Встроенный объект JavaScript `Date` включает лишь несколько встроенных форматов даты, и если они не удовлетворяют вашим потребностям,

то осуществить форматирование самостоятельно будет довольно сложно. К счастью, библиотека Moment.js хороша в этой области, и если вы требовательны к отображению даты, то я рекомендую использовать именно ее.

Для форматирования даты используйте метод format библиотеки Moment.js. Он получает строку из метасимволов, которые заменяются соответствующим компонентом даты. Например, строка "YYYY" будет заменена четырехразрядным годом. Вот несколько примеров форматирования даты встроенными методами объекта Date и более надежными методами Moment.js.

```
const d = new Date(Date.UTC(1930, 4, 10));

// здесь представлен вывод для Лос-Анджелеса

d.toLocaleDateString() // "5/9/1930"
d.toLocaleFormat()     // "5/9/1930 4:00:00 PM"
d.toLocaleTimeString() // "4:00:00 PM"
d.toTimeString()       // "17:00:00 GMT-0700 (Pacific Daylight Time)"
d.toUTCString()        // "Sat, 10 May 1930, 00:00:00 GMT"

moment(d).format("YYYY-MM-DD");              // "1930-05-09"
moment(d).format("YYYY-MM-DD HH:mm");        // "1930-05-09 17:00
moment(d).format("YYYY-MM-DD HH:mm Z");      // "1930-05-09 17:00 -07:00
moment(d).format("YYYY-MM-DD HH:mm [UTC]Z"); // "1930-05-09 17:00 UTC-07:00

moment(d).format("dddd, MMMM [the] Do, YYYY"); // "Friday, May the 9th, 1930"

moment(d).format("h:mm a");                  // "5:00 pm"
```

В этом примере продемонстрировано, насколько противоречивы и мало гибки встроенные возможности форматирования даты. К чести JavaScript следует заметить, что эти встроенные параметры форматирования действительно пытаются обеспечить формат, подходящий для региона пользователя. Если необходимо обеспечить форматирование даты в нескольких регионах, то это недорогой, хотя и не гибкий, способ сделать это.

Не будем приводить здесь полный справочник по опциям форматирования Moment.js; он доступен в сетевой документации. Достаточно будет сообщить, что, если у вас есть потребность в форматировании дат, то Moment.js почти наверняка поможет в этом. У нее есть некие общие соглашения по форматированию дат, подобные многим метаязыкам. Чем больше символов, тем подробнее, т.е. "M" даст 1, 2, 3...; "MMS" — 01, 02, 03...; "MMM" — Jan, Feb, Mar...; а "MMMM" — January, February, March.... Символ "o" в нижнем регистре обеспечит числительные: так "Do" даст 1st, 2nd, 3rd и т.д. Если вы хотите включить символы, которые не нужно интерпретировать как метасимволы, заключите их в квадратные скобки: "[M]M" даст M1, M2 и т.д.

Одно из затруднений, которые библиотека `Moment.js` не решает полностью, — это использование сокращений часового пояса, таких как EST или PST. Она исключила символ форматирования `z` в связи с отсутствием единых международных стандартов. Подробное обсуждение проблем с сокращениями часового пояса приведено в документации библиотеки `Moment.js`.

Компоненты даты

Если необходимо получить доступ к индивидуальным компонентам экземпляра `Date`, используйте соответствующие методы.

```
const d = new Date(Date.UTC(1815, 9, 10));

// здесь представлен вывод для Лос-Анджелеса
d.getFullYear()     // 1815
d.getMonth()        // 9 - October
d.getDate()         // 9
d.getDay()          // 1 - Monday
d.getHours()        // 17
d.getMinutes()      // 0
d.getSeconds()      // 0
d.getMilliseconds() // 0

// есть также эквиваленты UTC для вышеупомянутого:
d.getUTCFullYear()  // 1815
d.getUTCMonth()     // 9 - October
d.getUTCDate()      // 10
// ...и т.д.
```

Если вы будете использовать `Moment.js`, то вам вряд ли потребуется работать с индивидуальными компонентами, но нужно знать, что это возможно.

Сравнение дат

Для простых сравнений даты (действительно ли дата A следует после даты B или наоборот?) вы можете использовать встроенные операторы сравнения JavaScript. Помните, что экземпляры `Date` хранят дату как число, поэтому операторы сравнения просто работают с числами.

```
const d1 = new Date(1996, 2, 1);
const d2 = new Date(2009, 4, 27);

d1 > d2    // false
d1 < d2    // true
```

Арифметические операции с датами

Поскольку даты — это только числа, вы можете вычитать их для получения количества миллисекунд между ними.

```
const msDiff = d2 - d1;                // 417740400000 миллисекунд
const daysDiff = msDiff/1000/60/60/24; // 4834.96 дней
```

Это свойство облегчает также сортировку дат с использованием `Array.prototype.sort`.

```
const dates = [];
// создать несколько случайных дат
const min = new Date(2017, 0, 1).valueOf();
const delta = new Date(2020, 0, 1).valueOf() - min;
for(let i=0; i<10; i++)
    dates.push(new Date(min + delta*Math.random()));
// даты случайны и (вероятно) перемешаны
// мы можем отсортировать их (по убыванию):
dates.sort((a, b) => b - a);
// или возрастанию:
dates.sort((a, b) => a - b);
```

Библиотека `Moment.js` предоставляет множество мощных методов для общих операций с датами, позволяя добавлять или вычитать произвольные единицы времени.

```
const m = moment();       // сейчас
m.add(3, 'days');         // теперь m на три дня в будущем
m.subtract(2, 'years');   // теперь m на два года минус три дня в прошлом

m = moment();             // сброс
m.startOf('year');        // теперь m 1 января этого года
m.endOf('month');         // теперь m 31 января этого года
```

Библиотека `Moment.js` позволяет также сцеплять методы.

```
const m = moment()
    .add(10, 'hours')
    .subtract(3, 'days')
    .endOf('month');

// m - конец месяца, в котором вы оказались бы, если бы путешествовали
// 10 часов в будущее, а затем 3 дня назад
```

Удобные относительные даты

Довольно часто возникает необходимость представить информацию даты в относительном виде: не как конкретную дату, а "три дня назад". Библиотека `Moment.js` позволяет сделать это.

```
moment().subtract(10, 'seconds').fromNow();  // несколько секунд назад
moment().subtract(44, 'seconds').fromNow();  // несколько секунд назад
moment().subtract(45, 'seconds').fromNow();  // минуту назад
moment().subtract(5, 'minutes').fromNOw();   // 5 минут назад
moment().subtract(44, 'minutes').fromNOw();  // 44 минуты назад
moment().subtract(45, 'minutes').fromNOw();  // час назад
moment().subtract(5, 'hours').fromNOw();     // 4 часа назад
moment().subtract(21, 'hours').fromNOw();    // 21 час назад
moment().subtract(22, 'hours').fromNOw();    // день назад
moment().subtract(344, 'days').fromNOw();    // 344 дня назад
moment().subtract(345, 'days').fromNOw();    // год назад
```

Как можно заметить, в `Moment.js` выбраны некие произвольные (но разумные) контрольные точки для перехода к отображению других единиц. Это удобный способ получения относительных дат.

Заключение

Из этой главы вы должны были извлечь следующие уроки.

- Внутренне даты представляются как количество миллисекунд от Эпохи Unix (1 января 1970 года UTC).
- Создавая даты, помните о часовом поясе.
- Если необходимо сложное форматирование даты, используйте `Moment.js`.

В большинстве реальных приложений трудно избежать работы с датами. Мы надеемся, что эта глава дала вам понимание важнейших концепций. Полная и подробная документация по `Moment.js` содержится в сети разработчика Mozilla Developer Network.

Объект Math

В этой главе описан встроенный объект JavaScript `Math`, который содержит математические функции, обычно встречающиеся при разработке приложений (если вы осуществляете сложный математический анализ, вам, вероятно, имеет смысл воспользоваться библиотеками стороннего производителя).

Прежде чем углубляться в библиотеки, давайте вспомним, как JavaScript обрабатывает числа. В частности, вспомним что нет никакого специального целочисленного класса; все числа представляются как 64-битовые числа с плавающей запятой стандарта IEEE 754. Это упрощает задачу большинству функций в математической библиотеке: число есть число. Хотя никакой компьютер никогда не сможет точно представить произвольное вещественное число, с практической точки зрения вы можете считать числа JavaScript вещественными. Обратите внимание, что никакой встроенной поддержки для комплексных чисел в JavaScript нет. Если необходимы комплексные числа, сверхбольшие числа, более сложные структуры или алгоритмы, я рекомендую использовать библиотеку `Math.js`.

Кроме некоторых основ, эта глава не о математике. Этой теме посвящено множество других книг.

Для указания на то, что данное значение приблизительно, в комментариях к коду этой главы я буду использовать тильду (~) как префикс. Я также буду именовать свойства объекта `Math` *функциями*, а не *методами*. Хотя технически они являются статическими методами, различие здесь является чисто академическим, поскольку объект `Math` предоставляет пространство имен, а не контекст.

Форматирование чисел

Обычно числа необходимо форматировать, т.е. вместо того чтобы отображать `2.0093`, вы хотите отобразить `2.1` или вместо `1949032` вы хотите отобразить `1,949,032`.[1]

[1] В некоторых региональных форматах в качестве разделителей тысяч используются точки, а запятые используются как десятичный разделитель, в отличие от того, к чему, возможно, привыкли вы.

Хотя встроенная поддержка форматирования чисел в JavaScript весьма ограничена, в нее включена возможность отображения чисел с фиксированным количеством десятичных цифр, фиксированной точности и экспоненциальной формы записи. Кроме того, есть поддержка для отображения чисел с другим основанием, таким как двоичное, восьмеричное и шестнадцатеричное.

Все методы форматирования чисел в JavaScript возвращают строку, а не число. Дело в том, что только строка способна сохранить желаемое форматирование (при необходимости ее довольно просто преобразовать обратно в число). Поэтому числа следует форматировать лишь непосредственно перед их отображением; пока вы храните их или используете в вычислениях, числа должны оставаться неотформатированными.

Числа с фиксированным количеством десятичных цифр

Если необходимо отобразить фиксированное количество цифр после десятичной точки, вы можете использовать `Number.prototype.toFixed`.

```
const x = 19.51;
x.toFixed(3);          // "19.510"
x.toFixed(2);          // "19.51"
x.toFixed(1);          // "19.5"
x.toFixed(0);          // "20"
```

Обратите внимание, что это не усечение: вывод округляется до указанного количества десятичных цифр.

Экспоненциальная форма записи

Если необходимо отображать числа в экспоненциальной форме, используйте `Number.prototype.toExponential`.

```
const x = 3800.5;
x.toExponential(4);    // "3.8005e+4";
x.toExponential(3);    // "3.801e+4";
x.toExponential(2);    // "3.80e+4";
x.toExponential(1);    // "3.8e+4";
x.toExponential(0);    // "4e+4";
```

Подобно `Number.prototype.toFixed`, вывод округляется, а не усекается. Задается точность — количество цифр после десятичной точки.

Фиксированная точность

Если необходимо фиксированное количество цифр независимо от положения десятичной точки, вы можете использовать `Number.prototype.toPrecision`.

```
let x = 1000;
x.toPrecision(5);       // "1000.0"
x.toPrecision(4);       // "1000"
x.toPrecision(3);       // "1.00e+3"
x.toPrecision(2);       // "1.0e+3"
x.toPrecision(1);       // "1e+3"
x = 15.335;
x.toPrecision(6);       // "15.3350"
x.toPrecision(5);       // "15.335"
x.toPrecision(4);       // "15.34"
x.toPrecision(3);       // "15.3"
x.toPrecision(2);       // "15"
x.toPrecision(1);       // "2e+1"
```

Вывод округляется, и всегда будет иметь заданное количество цифр точности. При необходимости вывод может быть в экспоненциальной форме записи.

Другие основания

Если вы хотите отображать числа с другим основанием (двоичным, восьмеричным или шестнадцатеричным), используйте `Number.prototype.toString`, которому передается аргумент, определяющий основание (в диапазоне 2–36).

```
const x = 12;
x.toString();           // "12" (по основанию 10)
x.toString(10);         // "12" (по основанию 10)
x.toString(16);         // "c" (шестнадцатеричный)
x.toString(8);          // "14" (восьмеричный)
x.toString(2);          // "1100" (двоичный)
```

Дополнительное форматирование чисел

Если вы отображаете в своем приложении много чисел, ваши потребности могут быстро превзойти возможности встроенных методов JavaScript. Обычно необходимо следующее.

- Разделители тысяч.
- Иной способ отображения отрицательных чисел (например, с круглыми скобками).
- Инженерная форма записи (подобная экспоненциальной форме).
- Префиксы системы Си (мульти-, микро-, кило-, мега- и т.д.).

Обеспечение этих функциональных возможностей может быть хорошим самостоятельным упражнением. Но если особого желания нет, я рекомендую пользоваться *библиотекой* `Numeral.js`, которая предоставляет все эти и многие другие функциональные возможности.

Константы

Наиболее важные константы доступны как свойства объекта `Math`.

```
// фундаментальные константы
Math.E   // основание натурального логарифма: ~2.718
Math.PI  // отношение длины окружности к диаметру: ~3.142

// логарифмические константы доступны в библиотеках;
// подобные вызовы достаточно общеприняты и гарантируют удобство
Math.LN2     // натуральный логарифм 2: ~0.693
Math.LN10    // натуральный логарифм 10: ~2.303
Math.LOG2E   // логарифм по основанию 2 от Math.E: ~1.433
Math.LOG10E  // логарифм по основанию 10 от Math.E: 0.434

// алгебраические константы
Math.SQRT1_2 // корень квадратный из 1/2: ~0.707
Math.SQRT2   // корень квадратный из 2: ~1.414
```

Алгебраические функции

Возведение в степень

Базовая функция возведения в степень — это `Math.pow`, но есть и дополнительные функции для квадратного корня, кубического корня и степеней числа *e*, как показано в табл. 16.1.

Таблица 16.1. Функции возведения в степень

Функция	Описание	Примеры
`Math.pow(x, y)`	x^y	`Math.pow(2, 3)` `// 8` `Math.pow(1.7, 2.3)` `// ~3.39`
`Math.sqrt(x)` x	$\sqrt{x^2}$. Эквивалент `Math.pow(x, 0.5)`	`Math.sqrt(16)` `// 4` `Math.sqrt(15.5)` `// ~3.94`
`Math.cbrt(x)`	Кубический корень x. Эквивалент `Math.pow(x, 1/3)`	`Math.cbrt(27)` `// 3` `Math.cbrt(22)` `// ~2.8`
`Math.exp(x)`	e^x. Эквивалент `Math.pow(Math.E, x)`	`Math.exp(1)` `// ~2.718` `Math.exp(5.5)` `// ~244.7`

Функция	Описание	Примеры
`Math.expm1(x)`	e^x–1. Эквивалент `Math.exp(x) - 1`	`Math.expm1(1)` // ~1.718 `Math.expm1(5.5)` // ~243.7
`Math.hypot(x1, x2,...)`	Квадратный корень суммы аргументов: $\sqrt{x_1^2 + x_2^2 + \ldots}$	`Math.hypot(3, 4)` // 5 `Math.hypot(2, 3, 4)` // ~5.36

Логарифмические функции

Базовая функция натурального логарифма — `Math.log`. В некоторых языках под "log" понимают "логарифм по основанию 10", а под "ln" — "натуральный логарифм", поэтому имейте в виду, что в JavaScript "log" означает "натуральный логарифм". В спецификацию ES6 введена для удобства функция `Math.log10`.

Таблица 16.2. Логарифмические функции

Функция	Описание	Примеры
`Math.log(x)`	Натуральный логарифм `x`	`Math.log(Math.E)` // 1 `Math.log(17.5)` // ~2.86
`Math.log10(x)`	Логарифм по основанию 10 от `x`. Эквивалент `Math.log(x) / Math.log(10)`	`Math.log10(10)` // 1 `Math.log10(16.7)` // ~1.22
`Math.log2(x)`	Логарифм по основанию 2 от `x`. Эквивалент `Math.log(x) / Math.log(2)`	`Math.log2(2)` // 1 `Math.log2(5)` // ~2.32
`Math.log1p(x)`	Натуральный логарифм `1 + x`. Эквивалент `Math.log(1 + x)`	`Math.log1p(Math.E - 1)` // 1 `Math.log1p(17.5)` // ~2.92

Другое

В табл. 16.3 приведены другие числовые функции, которые позволяют вам выполнять такие популярные операции, как поиск абсолютного значения, наименьшего и наибольшего целого числа, а также знака числа, минимального или максимального числа в списке.

Таблица 16.3. Другие алгебраические функции

Функция	Описание	Примеры
`Math.abs(x)`	Абсолютное значение `x`	`Math.abs(-5.5)` // 5.5 `Math.abs(5.5)` // 5.5
`Math.sign(x)`	Знак `x`: если `x` — отрицательное, то `-1`; если `x` — положительное, то `1`; а если `x` — `0`, то `0`	`Math.sign(-10.5)` // -1 `Math.sign(6.77)` // 1

Функция	Описание	Примеры	
`Math.ceil(x)`	Наименьшее целое число, большее или равное `x`	`Math.ceil(2.2)` `Math.ceil(-3.8)`	`// 3` `// -3`
`Math.floor(x)`	Наибольшее целое число, меньше или равное `x`	`Math.floor(2.8)` `Math.floor(-3.2)`	`// 2` `// -4`
`Math.trunc(x)`	Целая часть `x` (все десятичные цифры удаленны)	`Math.trunc(7.7)` `Math.trunc(-5.8)`	`// 7` `// -5`

`Math.round(x)`	Округляет `x` до ближайшего целого числа	`Math.round(7.2)`
		`Math.round(7.7)`
		`Math.round(-7.7)`
		`Math.round(-7.2)`
			`// 7`
			`// 8`
			`// -8`
			`// -7`

`Math.min (x1, x2,...)`	Возвращает минимальный аргумент	`Math.min(1, 2)`
		`Math.min(3, 0.5, 0.66)`
		`Math.min(3, 0.5, -0.66`
			`// 1`
			`// 0.5`
			`// -0.66`

`Math.max (x1, x2,...)`	Возвращает максимальный аргумент	`Math.max(1, 2)`
		`Math.max(3, 0.5, 0.66)`
		`Math.max(-3, 0.5, -0.66)`
			`// 2`
			`// 3`
			`// 0.5`

Генерация псевдослучайных чисел

Генерацию псевдослучайных чисел обеспечивает `Math.random`. Она возвращает псевдослучайное число, большее или равное 0 и меньшее 1. Из уроков по алгебре вы, возможно, помните, что диапазоны чисел зачастую обозначаются с использованием квадратных скобок (включительно) и круглых скобок (исключительно). С учетом этого `Math.random` возвращает числа в диапазоне [0, 1).

`Math.random` не предоставляет методов для создания псевдослучайных чисел в разных диапазонах. Некоторые из наиболее популярных формул для получения других диапазонов приведены в табл. 16.4. В этой таблице `x` и `y` обозначают вещественные числа, `m` и `n` — целые числа.

Таблица 16.4. Генерация псевдослучайных чисел

Диапазон	Пример
[0, 1)	`Math.random()`
[x, y)	`x + (y-x)*Math.random()`
Целое в [m, n)	`m + Math.floor((n-m)*Math.random())`
Целое в [m, n]	`m + Math.floor((n-m+1)*Math.random())`

Генератор псевдослучайных чисел JavaScript зачастую ругают за невозможность задать *начальное значение*, что очень важно при тестировании некоторых алгоритмов, в которых задействованы псевдослучайные числа. Если необходимы генераторы псевдослучайных чисел с начальными значениями, используйте *пакет* `seedrandom.js` Дэвида Бау (David Bau).

Генераторы псевдослучайных чисел (Pseudorandom Number Generator — PRNG) очень часто (но неправильно) называют просто "генераторами случайных чисел" (Random Number Generator — RNG). PRNG генерируют числа, которые в большинстве практических случаев вполне равнозначны случайным, но генерация истинно случайного числа — весьма трудная задача.

Тригонометрические функции

Здесь нет никаких неожиданностей. Доступны синус, косинус, тангенс и их обратные значения, как показано в табл. 16.5. Все тригонометрические функции в библиотеке `Math` работают в радианах, а не градусах.

Таблица 16.5. Тригонометрические функции

Функция	Описание	Примеры	
`Math.sin(x)`	Синус `x` в радианах	`Math.sin(Math.PI/2)` `Math.sin(Math.PI/4)`	`// 1` `// ~0.707`
`Math.cos(x)`	Косинус `x` в радианах	`Math.cos(Math.PI)` `Math.cos(Math.PI/4)`	`// -1` `// ~0.707`
`Math.tan(x)`	Тангенс `x` в радианах	`Math.tan(Math.PI/4)` `Math.tan(0)`	`// ~1` `// 0`
`Math.asin(x)`	Обратный синус (арксинус) `x` (в радианах)	`Math.asin(0)` `Math.asin(Math.SQRT1_2)`	`// 0` `// ~0.785`
`Math.acos(x)`	Обратный косинус (арккосинус) `x` (в радианах)	`Math.acos(0)` `Math.acos(Math.SQRT1_2)`	`//~1.57+` `// ~0.785+`
`Math.atan(x)`	Обратный тангенс (арктангенс) `x` (в радианах)	`Math.atan(0)` `Math.atan(Math.SQRT1_2)`	`// 0` `// ~0.615`
`Math.atan2 (y, x0)`	Угол (в радианах) от оси `x` к точке `(x, y)`, отсчитанный против часовой стрелки	`Math.atan2(0, 1)` `Math.atan2(1, 1)`	`// 0` `// ~0.785`

Если понадобится иметь дело с градусами, то сначала их нужно преобразовать в радианы. Формула очень проста: разделите число градусов на 180 и умножьте на π. Для этого довольно просто написать вспомогательные функции.

```
function deg2rad(d) { return d/180*Math.PI; }
function rad2deg(r) { return r/Math.PI*180; }
```

Гиперболические функции

Гиперболические функции, подобно тригонометрическим, вполне стандартны, как можно заметить в табл. 16.6.

Таблица 16.6. Гиперболические функции

Функция	Описание	Примеры
`Math.sinh(x)`	Гиперболический синус x	`Math.sinh(0)` // *0* `Math.sinh(1)` // *~1.18*
`Math.cosh(x)`	Гиперболический косинус x	`Math.cosh(0)` // *1* `Math.cosh(1)` // *~1.54*
`Math.tanh(x)`	Гиперболический тангенс x	`Math.tanh(0)` // *0* `Math.tanh(1)` // *~0.762*
`Math.asinh(x)`	Обратный гиперболический синус (ареасинус) x	`Math.asinh(0)` // *0* `Math.asinh(1)` // *~0.881*
`Math.acosh(x)`	Обратный гиперболический косинус (ареакосинус) x	`Math.acosh(0)` // *NaN* `Math.acosh(1)` // *0*
`Math.atanh(x)`	Обратный гиперболический тангенс (ареатангенс) x	`Math.atanh(0)` // *0* `Math.atanh(0)` // *~0.615*

Регулярные выражения

Регулярные выражения (regular expression) позволяют выполнить сложный функциональный анализ текстовых строк. Если вам нужно проанализировать строки, которые выглядят, как адрес электронной почты или URL, или номер телефона, вот здесь и пригодятся регулярные выражения. Естественным дополнением к процессу анализа строк является операция замены строк, разумеется с помощью регулярных выражений. Например, может возникнуть необходимость определить строки, которые выглядят, как адреса электронной почты, и заменить их гиперссылками на эти адреса.

Во многих вводных курсах по регулярным выражениям используются такие загадочные примеры, как "поиск соответствия *aaaba* и *abaaba*, но не *abba*". С их помощью демонстрируется преимущество разделения сложных регулярных выражений на четкие блоки функциональных возможностей. Однако недостаток таких примеров лежит на поверхности — это их бессмысленность (когда вам в реальности понадобится искать соответствие строке *aaaba*?). Я собираюсь познакомить вас с возможностями регулярных выражений на реальных и практических примерах.

Термин "регулярное выражение" (regular expression) зачастую сокращают до "regex" или "regexp"; в этой книге мы будем использовать их для краткости.

Распознавание и замена подстрок

Основная задача регулярного выражения — распознать подстроку в пределах строки и, при необходимости, заменить ее на другую строку. Регулярные выражения позволяют делать это с невероятной мощью и гибкостью. Таким образом, прежде чем перейти к ним, давайте сначала кратко рассмотрим процедуру поиска подстроки без использования регулярного выражения и ее замену с помощью методов `String.prototype`. Такая задача возникает довольно часто даже в небольших приложениях.

Если все, что необходимо сделать, — это определить, существует ли некая подстрока в строке, будет вполне достаточно следующих методов `String.prototype`.

```
const input = "As I was going to Saint Ives";
input.startsWith("As")       // true
input.endsWith("Ives")       // true
```

```
input.startsWith("going", 9) // true -- начать 9-й позиции
input.endsWith("going", 14)  // true -- считать 14-ю позицию концом строки
input.includes("going")      // true
input.includes("going", 10)  // false -- начать с 10-й позиции
input.indexOf("going")       // 9
input.indexOf("going", 10)   // -1
input.indexOf("nope")        // -1
```

Обратите внимание, что все эти методы чувствительны к регистру. Так, вызов `input.startsWith("as")` возвратил бы `false`. Если вы хотите сделать сравнение независящим от регистра, можете просто преобразовать строку в нижний регистр.

```
input.toLowerCase().startsWith("as") // true
```

Обратите внимание, что это не изменяет исходную строку; `String.prototype.toLowerCase` возвращает новую строку и не изменяет исходную (не забывайте, что строки в JavaScript неизменны!).

Если мы хотим сделать следующий шаг и, найдя подстроку, заменить ее, то можем использовать `String.prototype.replace`.

```
const input = "As I was going to Saint Ives";
const output = input.replace("going", "walking");
```

И снова исходная строка (`input`) не изменяется в результате выполнения этой замены; `output` теперь содержит новую строку, в которой слово `"going"` заменено на `"walking"` (если вам действительно нужно изменить значение переменной `input`, воспользуйтесь оператором присваивания).

Создание регулярных выражений

Прежде чем перейти к сложности метаязыка регулярных выражений, давайте поговорим о том, как они фактически создаются и используются в JavaScript. В этих примерах, как и прежде, мы будем искать определенную строку. Хотя это и слишком легкая задача для регулярных выражений, так проще объяснить, как они используются.

Регулярные выражения в JavaScript представлены классом `RegExp`. Хотя вы можете создать регулярные выражения, используя конструктор `RegExp`, они настолько важны, что для них предусмотрен собственный литеральный синтаксис. Литералы регулярных выражений заключаются в символы наклонной черты.

```
const re1 = /going/;              // регулярное выражение, способное
                                  // искать слово "going"
const re2 = new RegExp("going"); // эквивалентный конструктор объекта
```

В использовании конструктора `RegExp` есть определенный резон, который мы рассмотрим далее в этой главе, но, кроме этого частного случая, следует предпочесть более удобный литеральный синтаксис.

Поиск с использованием регулярных выражений

После создания регулярного выражения его можно использовать для поиска подстроки в строке несколькими способами.

Чтобы понять опции для замены, сначала рассмотрим небольшой пример применения метаязыка регулярного выражения (использование статической строки здесь было бы очень скучным). Мы используем регулярное выражение `/\w{3,}/ig`, которому будут соответствовать все слова из трех или более символов (без учета регистра). Не волнуйтесь о понимании всего прямо сейчас; это придет далее в этой главе. Теперь мы можем рассмотреть доступные нам методы поиска.

```
const input = "As I was going to Saint Ives";
const re = /\w{3,}/ig;

// начнем со строки (input)
input.match(re);  // ["was", "going", "Saint", "Ives"]
input.search(re); // 5 (первое трехбуквенное слово начинается с индекса 5)

// начнем с регулярного выражения (re)
re.test(input);   // true (input содержит по крайней мере одно
                  // трехбуквенное слово)
re.exec(input);   // ["was"] (первое соответствие)
re.exec(input);   // ["going"] (exec "помнит", где он находится)
re.exec(input);   // ["Saint"]
re.exec(input);   // ["Ives"]
re.exec(input);   // null -- соответствий больше нет

// обратите внимание, что любой из этих методов применяется
// непосредственно с литералом регулярного выражения
input.match(/\w{3,}/ig);
input.search(/\w{3,}/ig);
/\w{3,}/ig.test(input);
/\w{3,}/ig.exec(input);
// ...
```

Из всех перечисленных выше методов `RegExp.prototype.exec` предоставляет больше всего информации, но на практике почему-то он используется очень редко. Я чаще всего использую `String.prototype.match` и `RegExp.prototype.test`.

Замена с использованием регулярных выражений

Мы уже видели применение метода `String.prototype.replace` для замены простых строк. Однако он также позволяет использовать регулярные выражения и способен на гораздо большее! Давайте начнем с простого примера — замены всех слов, состоящих из 4-х и более символов.

```
const input = "As I was going to Saint Ives";
const output = input.replace(/\w{4,}/ig, '****'); // "As I was ****
                                                  // to **** ****"
```

О намного более сложных методах замены мы узнаем далее в этой главе.

Переработка входных данных

В самом простом случае регулярные выражения можно считать "средством поиска подстрок в пределах строки" (такая задача зачастую образно называется "поиском иголки в стоге сена"), и обычно это все, что требуется программисту. Однако ограничивая себя такими рамками вы будете не в состоянии понять истинную природу регулярных выражений и оценить всю их мощь для решения более сложных задач.

Регулярное выражение можно также считать *шаблоном для переработки входных строк*. Побочным продуктом этого подхода становится процесс поиска соответствия строк.

Чтобы лучше осмыслить работу регулярных выражений, стоит воспользоваться аналогией с популярной детской игрой, в которой вам предлагается выделить слова в таблице букв. Мы будем игнорировать диагональные и вертикальные соответствия; фактически давайте рассматривать только первую строку в этой словесной игре.

```
X J A N L I O N A T U R E J X E E L N P
```

Люди очень хорошо играют в эту игру. Мы можем посмотреть и сразу же найти слова `LION`, `NATURE` и `EEL` (а также `ION`, раз уж такое дело). Компьютеры, и регулярные выражения, не столь умны. Давайте рассмотрим регулярные выражения на примере этой игры. Это позволить нам не только увидеть, как они работают, но и ознакомиться с ограничениями, о которых необходимо знать.

Чтобы не усложнять, укажем обработчику регулярного выражения, что мы ищем слова `LION`, `ION`, `NATURE` и `EEL`. Другими словами, мы дадим ему готовые ответы и посмотрим, сможет ли он их найти.

Обработчик регулярного выражения начинает свой анализ с первого символа строки — `X`. Он определяет, что ни одно из искомых слов не начинается с символа `X` и поэтому сигнализирует об "отсутствии соответствия". Однако на этом его работа не заканчивается, и он начинает обработку следующего символа, `J`. Здесь ситуация та же. Затем переходит к символу `A`. После продвижения просмотренные обработчиком регулярного выражения символы считаются *переработанными* (consumed). Ничего интересного не происходит, пока ему не встретится буква `L`. Обработчик регулярного выражения замечает "Ага, это может быть слово `LION`!" Поскольку с этого символа может начаться соответствие заданным словам, обработчик *не перерабатывает символ* `L`. Это важный момент для понимания. Обработчик движется дальше, встречая букву `I`, затем `O`, затем `N`. Теперь он распознает соответствие. Это успех! Распознав соответствие, можно считать переработанным целое слово, таким образом, символы `L`, `I`, `O` и `N` теперь относятся к переработанным. А дальше все становится еще интереснее. Слова `LION` и `NATURE`

перекрываются. Для людей это не составляет особой проблемы. Но для обработчика регулярного выражения это очень серьезное препятствие, он не рассматривает то, что уже было переработано. Таким образом, обработчик не "возвращается" на шаг назад, чтобы попытаться найти соответствие среди переработанных символов. Поэтому, он не найдет слово NATURE, поскольку буква N уже была переработана. Все, что он может найти, — это строка ATURE, которая не относится ни к одному из искомых слов. Однако в конечном счете обработчик найдет слово EEL.

Давайте снова вернемся к примеру и в слове LION заменим O на X. Что будет теперь? Когда обработчик регулярного выражения дойдет до буквы L, он снова распознает возможное соответствие слову LION, а потому не переработает букву L. Он перейдет к букве I, также не перерабатывая ее. Затем он доходит до буквы X и понимает, что соответствия нет: ему не поставлена задача искать слова, которое начинаются с LIX. Теперь обработчик регулярного выражения возвращается туда, где, как предполагалось, должно было быть соответствие (к букве L), перерабатывает все буквы вплоть до X и переходит к обработке следующей по порядку буквы обычным образом. В данном случае он найдет соответствие слову NATURE, поскольку символ N не был переработан как часть слова LION.

Часть этого процесса представлена на рис. 17.1.

...продолжаем, пока не будет переработана вся входная строка

Рис. 17.1. Пример обработки регулярного выражения

Прежде чем мы перейдем к обсуждению специфических особенностей метаязыка регулярных выражений, давайте абстрактно рассмотрим алгоритм "переработки" строки обработчиком регулярного выражения.

- Строки перерабатываются слева направо.
- После переработки символа к нему больше не возвращаются.
- При отсутствии соответствия обработчик регулярного выражения переходит на один символ вперед и делает следующую попытку найти соответствие.
- Если соответствие найдено, обработчик регулярного выражения перерабатывает все найденные символы сразу; поиск соответствия продолжается со следующего символа (если регулярное выражение глобально, о чем речь пойдет позже).

Это общий алгоритм, и нет ничего удивительного в том, что подробности окажутся намного сложнее. В частности, алгоритм может быть прерван, если обработчик регулярного выражения посчитает, что соответствий больше не будет.

Когда мы перейдем к специфическим особенностям метаязыка регулярных выражений, постарайтесь вспомнить этот алгоритм; представьте, что ваши строки перерабатываются слева направо, по одному символу за раз, пока не встретится соответствие, и тогда перерабатываются сразу все найденные символы.

Чередование

Предположим, у вас есть HTML-страница, сохраненная в строковой переменной, и вы хотите найти все дескрипторы, способные ссылаться на внешний ресурс (`<a>`, `<area>`, `<link>`, `<script>`, `<source>`, а иногда и `<meta>`). Кроме того, некоторые из дескрипторов могут быть записаны в смешанном регистре (`<Area>`, `<LINKS>` и т.д.). Для решения этой задачи с помощью регулярных выражений применяют *чередование* (alternation).

```
const html = 'HTML with <a href="/one">one link</a>, and some JavaScript.' +
    '<script src="stuff.js"></script>';
const matches = html.match(/area|a|link|script|source/ig); // первая попытка
```

Вертикальная черта (`|`) является метасимволом регулярного выражения, что означает чередование. Суффикс `ig` указывает игнорировать регистр (`i`) и искать глобально (`g`). Без опции `g` будет возвращено только первое соответствие. Это читалось бы так: "Найти все варианты текста `area`, `a`, `link`, `script` или `source`, игнорируя регистр". Проницательный читатель мог бы задаться вопросом "Почему мы поместили `area` перед `a`?" Это связано с тем, что обработчик регулярного выражения выполняет операцию чередования слева направо. Другими словами, если в строке есть дескриптор `area`, то будет найдено только соответствие букве `a` и произойдет переход. Затем символ `a` перерабатывается, а строка `rea` не соответствует ничему искомому. Таким

образом, сначала следует найти соответствие area, а затем a; в противном случае соответствия area никогда не будет найдено!

Запустив этот пример, вы обнаружите, что получилось несколько непреднамеренных соответствий: слово link внутри дескриптора <a>, а также все экземпляры символа a, которые *не входят* в HTML-дескрипторы, а просто являются частью английского текста. Один из способов решения этой проблемы подразумевал бы изменение регулярного выражения на /<area|<a|<link|<script|<source/ (угловые скобки не являются метасимволом регулярного выражения, поэтому их не нужно экранировать), но мы сделаем все намного лучше.

Анализ HTML-кода

В предыдущем примере мы выполнили вполне обычную задачу с регулярными выражениями: нашли соответствие заданным строкам в HTML-коде. Хотя эта задача вполне традиционна, я должен предупредить вас, что несмотря на то, что, используя регулярные выражения, вы можете делать весьма полезные вещи с HTML-кодом, вы не можете *анализировать* его с помощью регулярных выражений. *Анализ* (parsing) подразумевает полное разделение чего-то на составные части. Обработчик регулярного выражения способен анализировать текст только *обычных* (regular) языков (отсюда и название!). Структура обычных языков чрезвычайно проста, но чаще всего вы будете использовать регулярные выражения для обработки более сложных языков. Зачем это предупреждение, если регулярные выражения применимы для более сложных языков? Важно понимать, что регулярные выражения имеют ограничения и в соответствующем случае лучше использовать нечто более мощное. Даже при том что мы будем использовать регулярные выражения для работы с HTML-кодом, вполне возможно создать такой код, который окажется не по силам для регулярных выражений. Чтобы получить решение, которое сработает в 100% случаев, вам нужно использовать *анализатор* (parser). Рассмотрим следующий пример.

```
const html = '<br> [!CDATA[[<br>]]';
const matches = html.match(/<br>/ig);
```

Это регулярное выражение найдет соответствие дважды; но в нашем примере есть только один истинный дескриптор
; другая соответствующая строка — это просто *символьные данные* (character data — CDATA), а не HTML-код. Возможности регулярных выражений также чрезвычайно ограничены, когда дело доходит до распознавания иерархических структур (таких, как дескриптор <a>, вложенный в дескриптор <p>). Теоретические объяснения этих ограничений не рассматриваются в данной книге, но вывод ясен: если вы изо всех сил пытаетесь создать регулярное выражение для распознавания чего-то очень сложного (такого, как языковые конструкции в HTML-коде), всегда имейте в виду, что регулярное выражение может просто не быть подходящим инструментом для этого.

Наборы символов

Наборы символов (character set) обеспечивают компактный способ записи чередования *одиночного символа.* Впоследствии мы скомбинируем их с возможностью повторения и увидим, как можно расширить регулярное выражение для распознавания нескольких символов. Например, если вам нужно найти все числа в строке, можно использовать чередование.

```
const beer99 = "99 bottles of beer on the wall " +
    "take 1 down and pass it around -- " +
    "98 bottles of beer on the wall.";
const matches = beer99.match(/0|1|2|3|4|5|6|7|8|9/g);
```

Как утомительно! А что если мы хотим распознавать не числа, а символы? Числа *и* символы? Наконец, что если вы хотите распознавать все, что не число? Вот где в игру вступают наборы символов. В самом простом случае они обеспечивают более компактный способ представления чередования одиночных цифр. Более того, они позволяют задавать *диапазоны.* Вот как мы могли бы переписать приведенное выше.

```
const m1 = beer99.match(/[0123456789]/g); // хорошо
const m2 = beer99.match(/[0-9]/g);        // еще лучше!
```

Диапазоны можно даже объединять. Вот как мы распознавали бы символы, числа и некоторые из знаков пунктуации (им соответствовать будет все в первоначальной строке, кроме пробельных символов).

```
const match = beer99.match(/[\-0-9a-z.]/ig);
```

Обратите внимание, что порядок следования здесь не имеет значения: мы легко могли бы использовать `/[.a-z0-9\-]/`. Для распознавания символа "-" его следует экранировать; в противном случае JavaScript попытается интерпретировать его как часть диапазона (вы можете также поместить его прямо перед закрывающей квадратной скобкой без экранирования).

Другое очень мощное свойство наборов символов — это способность *инвертировать* наборы символов. Инвертированные наборы символов говорят: "Соответствует все, *кроме* этих символов". Для инверсии набора символов используйте символ галочки (^) как первый символ в наборе.

```
const match = beer99.match(/[^\-0-9a-z.]/);
```

В нашей исходной строке этому регулярному выражению будет соответствовать только пробельный символ (если нам нужно распознать только пробельные символы, есть куда лучший способ сделать это, о чем мы узнаем вскоре).

Именованные наборы символов

Некоторые наборы символов настолько распространены (и настолько полезны), что для них есть удобные сокращения.

Именованный набор символов	Эквивалент	Примечание
`\d`	`[0-9]`	
`\D`	`[^0-9]`	
`\s`	`[\t\v\n\r]`	Включает символы табуляции, пробела и вертикальной табуляции
`\S`	`[^\t\v\n\r]`	
`\w`	`[a-zA-Z_]`	Обратите внимание, что черточки и точки сюда не входят, что делает этот набор неподходящим для распознавания классов CSS и имен доменов
`\W`	`[^a-zA-Z_]`	

Чаще всех из этих сокращений, вероятно, используется набор пробельных символов (`\s`). Например, пробелы часто используется для выравнивания кода, поэтому в процессе его анализа программным способом, вам, вероятно, понадобиться учитывать любое возможное количество пробельных символов.

```
const stuff =
    'hight:     9\n' +
    'medium:    5\n' +
    'low:       2\n';
const levels = stuff.match(/:\s*[0-9]/g);
```

(Символ `*` после `\s` означает "любое количество пробелов", о чем мы узнаем вскоре.)

Не упускайте ценность инвертированных символьных классов (`\D`, `\S` и `\W`); они представляют отличный способ избавления от нежелательного хлама. Например, прекрасная идея нормализовать номера телефонов перед их сохранением в базе данных. При вводе номеров телефона для их читаемости люди могут пользоваться разными разделителями: черточками, точками, круглыми скобками и пробелами. Но для поиска, использования в качестве ключей и идентификации было бы хорошо, если бы номера телефонов состояли только из десяти цифр (или больше, если мы говорим о международных номерах телефона). При использовании набора `\D` (т.е. все что не цифры) это сделать очень просто.

```
const messyPhone = '(505) 555-1515';
const neatPhone = messyPhone.replace(/\D/g, '');
```

Точно так же я часто использую набор `\S` для проверки того, что в обязательных полях ввода есть хоть какие-то данные (в них должен быть по крайней мере один символ, и это не должен быть пробел!).

```
const field = ' something ';
const valid = /\S/.test(field);
```

Повторение

Повторение (repetition) метасимволов позволяет задать количество повторов неких соответствий. Рассмотрим предыдущий пример, в котором мы распознавали одиночные цифры. Что если вместо этого мы должны распознавать *числа* (которые могут состоять из нескольких расположенных рядом цифр)? Мы могли бы использовать то, что уже знаем, и сделать так.

```
const match = beer99.match(/[0-9][0-9][0-9]|[0-9][0-9]|[0-9]/);
```

Обратите внимание, что здесь снова необходимо распознавать более специфические строки (трехзначные числа) прежде, чем менее специфические (двузначные числа). Это регулярное выражение будет работать для одно-, двух- и трехзначных чисел, но, можно распознавать и четырехзначные числа, просто добавив новый шаблон в начало нашего чередования. К счастью, есть лучший путь.

```
const match = beer99.match(/[0-9]+/);
```

Обратите внимание на знак + после символьной группы: он указывает, что *предыдущий элемент* должен соответствовать *один или несколько раз*. Фраза "Предыдущий элемент" зачастую сбивает новичков с толку. Метасимволы повторения — это *модификаторы* (modifier), которые модифицируют *то, что расположено перед ними*. Они ничего не делают самостоятельно и не могут этого делать по определению! Есть пять модификаторов повторения.

Модификатор	Описание	Пример
`{n}`	Точно `n` раз	`/d{5}/` соответствует только пятизначным числам (например, почтовым индексам)
`{n,}`	По крайней мере `n` раз	`/\d{5,}/` соответствует только числам с пятью и более цифрами
`{n, m}`	По крайней мере `n`, но не более `m` раз	`/\d{2,5}/` соответствует только числам, состоящим по крайней мере из двух, но не больше пяти цифр
`?`	Ноль или один раз. Эквивалент `{0,1}`	`/[a-z]\d?/i` соответствует буквам английского алфавита, за которыми *может* следовать (а может и нет!) одна цифра

Модификатор	Описание	Пример
*	Ноль или несколько раз (иногда называется "звездой Клини")	`/[a-z]\d*/i` соответствует буквам английского алфавита, за которыми *может* следовать (а может и нет!) сразу несколько цифр
+	Один или несколько	`/[a-z]\d+/i` соответствует буквам английского алфавита, за которыми *должна* следовать одна или несколько цифр

Метасимвол "точка" и экранирование

Точка в регулярном выражении — это специальный символ, который означает "соответствие чему-нибудь" (кроме символов перевода строки). Очень часто этот подходящий для всего метасимвол используется для переработки не интересующих частей входной строки. Давайте рассмотрим пример поиска во входной строке одного почтового индекса, состоящего из пяти цифр, при этом остальная часть строки нас не интересует.

```
const input = "Address: 333 Main St., Anywhere, NY, 55532. Phone: 555-555-2525.";
const match = input.match(/\d{5}.*/);
```

Очень часть возникает необходимость распознать в строке символ точки, встречающийся, например, в доменном имени или IP-адресе. Аналогично вам может понадобиться распознать символы, соответствующие метасимволам регулярного выражения, например, звездочки или круглые скобки. Для этой цели *любой* такой символ следует экранировать, т.е. поместить перед ним обратную косую черту.

```
const equation = "(2 + 3.5) * 7";
const match = equation.match(/\(\d \+ \d\.\d\) \* \d/);
```

Многие читатели наверняка помнят работу в командной строке с *шаблонами имен файлов*. Например, шаблон `*.txt` соответствовал именам "любых текстовых файлов". Здесь `*` — это метасимвол "шаблона", означающий, что он соответствует чему угодно. Если это вам знакомо, использование символа `*` в регулярных выражениях может смутить вас, поскольку здесь она означает нечто совсем иное и не может использоваться в качестве самостоятельного символа. Точка в регулярном выражении больше всего похожа на шаблон `*` в масках имен файлов, а точка в шаблоне имени файла соответствует только одному символу точки, а не произвольной строке, как в шаблоне регулярного выражения.

Шаблон, соответствующий всему

Поскольку точка в регулярном выражении соответствует любому символу, *кроме* символа перевода строки, как распознать абсолютно любые символы, *включая* и символы новых строк? (Эта задача встречается чаще, чем вы могли бы подумать!) Есть много способов сделать это, но, вероятно, самый распространенный — `[\s\S]`. Данный шаблон соответствует любому пробельному символу... и всему, что им не является. Короче говоря, абсолютно всему!

Группировка

Изученные до сих пор конструкции регулярных выражений позволяют выявлять только одиночные символы (хотя повторение позволяет распознавать повторяющиеся символы, но это все еще одиночные символы!). *Группировка* (grouping) позволяет создавать *подвыражения* (subexpression), которые затем можно обработать, как единый блок.

Кроме возможности создавать подвыражения, группировка позволяет "захватывать" результаты групп, чтобы использовать их позже. Такое поведение принято по умолчанию, однако есть способ создать "не захватываемую группу" (noncapturing group), с чего мы и начнем. Если вы уже пользовались раньше регулярными выражениями, это может показаться для вас новостью, но я рекомендую всегда использовать там где это возможно обычные не захватываемые группы. У них есть преимущество в производительности, поэтому если вы не собираетесь использовать результаты работы группы позже, то используйте обычные группы. Группы определяются с помощью круглых скобок, а обычная не захватываемая группа определяется как `(?:`*<подвыражение>*`)`, где *<подвыражение>* — это шаблон того, что вы пытаетесь распознать. Давайте рассмотрим несколько примеров. Предположим, что вам нужно распознать имена доменов верхнего уровня `.com`, `.org` и `.edu`.

```
const text = "Visit oreilly.com today!";
const match = text.match(/[a-z]+(?:\.com|\.org|\.edu)/i);
```

Еще одно преимущество групп заключается в том, что к ним можно применить повторение. Обычно повторение применяется только к *одиночному символу*, расположенному слева от метасимвола повторения. Группы позволяют применять повторение к целым строкам. Вот общий пример. Если вам нужно распознать URL, которые начинаются с `http://`, `https://` или просто с `//` (URL, независимые от протокола), можете использовать группу с ноль или одним (`?`) повторением.

```
const html = '<link rel="stylesheet" href="http://insecure.com/stuff.css">\n' +
  '<link rel="stylesheet" href="https://secure.com/securestuff.css">\n' +
  '<link rel="stylesheet" href="//anything.com/flexible.css">';

const matches = html.match(/(?:https?)?\/\/[a-z][a-z0-9-]+[a-z0-9]+/ig);
```

Выглядит малопонятно, как абракадабра, не так ли? Для меня тоже. Но в этом примере заключено столько мощи, что на нем стоит остановиться и рассмотреть его подробно. Начнем с не захватываемой группы: `(?:https?)?`. Обратите внимание на два метасимвола повторения "ноль или один". Первый говорит, что "`s` опционально". Помните, что символы повторения обычно относятся только к тому символу, который расположен непосредственно перед ними (т.е. слева от них). Второй относится ко всей *группе*, расположенной слева от него. Таким образом, все вместе это выражение будет распознавать пустую строку (нулевое количество `http?`), `http` или `https`. Далее мы распознаем две косые черты (обратите внимание, мы экранируем их обратной косой чертой: `\/\/`). Затем следует довольно сложный символьный класс. Вполне очевидно, что имена доменов могут содержать буквы и цифры, но в них также могут встречаться дефисы (но начинаться они должны с буквы и не могут заканчиваться дефисом).

Этот пример далеко не совершенен. Например, он распознал бы URL `//gotcha` (глюк) без домена верхнего уровня, точно так же, как и `//valid.com`. Однако распознавание полностью правильных URL — намного более сложная задача, решать которую пока что нам не зачем.

Если вы сыты по горло всеми этими недостатками ("наш пример распознает некорректные URL"), знайте, что вы не обязаны делать все и сразу! Фактически для решения подобной задачи при сканировании веб-сайтов я использую очень похожее регулярное выражение. Сначала я просто собираю *все* URL (или похожие на URL), а затем делаю вторую фазу анализа, чтобы отбросить недопустимые или нерабочие URL и т.п. Не увлекайтесь слишком созданием совершенных регулярных выражений, которые учитывают каждый вообразимый случай. Мало того что иногда это невозможно, но зачастую, даже когда теоретически возможно, требует ненужных усилий. Очевидно, иногда уместно или даже нужно рассматривать все варианты, например когда вы просеиваете пользовательский ввод для предотвращения атак методом инъекции. В таком случае вам понадобится проявить дополнительные усилия и сделать свое регулярное выражение "пуленепробиваемым".

Ленивое и жадное распознавания

Что отличает дилетантов в регулярных выражениях от профессионалов — так это понимание преимуществ ленивых соответствий перед жадными. Регулярные выражения стандартно являются *жадными* (greedy), т.е. они будут распознавать максимально возможную строку перед остановкой. Рассмотрим классический пример.

Есть некий HTML-код, и вы хотите заменить в нем, например, дескриптор дескриптором . Вот наша первая попытка.

```
const input = "Regex pros know the difference between\n" +
    "<i>greedy</i> and <i>lazy</i> matching.";
input.replace(/<i>(.*)<\/i>/ig, '<strong>$1</strong>');
```

Часть $1 в строке замены будет заменена содержимым группы (.*) в регулярном выражении (подробности — далее).

Давайте опробуем это. Вы получите следующий неутешительный результат.

```
"Regex pros know the difference between
<strong>greedy</i> and <i>lazy</strong> matching."
```

Чтобы понять происходящее здесь, давайте вспомним, как функционирует обработчик регулярного выражения: он перерабатывает входные данные, пока не найдет соответствие, а затем снова продолжает переработку. Стандартно он делает это *жадным* способом: находит первое , а затем говорит "Я не остановлюсь, пока не увижу *самый последний* . Поскольку есть два экземпляра , он закончит свою работу на втором, а не на первом.

Есть несколько способов исправить этот пример, но поскольку мы обсуждаем различие между жадным и ленивым распознаваниями, давайте создадим ленивый метасимвол повторения (*). Для этого нужно просто поместить после него вопросительный знак.

```
input.replace(/<i>(.*?)<\/i>/ig, '<strong>$1</strong>');
```

Регулярное выражение — точно то же, за исключением вопросительного знака после метасимвола *. Теперь процессор регулярного выражения рассматривает его так: "я остановлюсь, как только увижу первый ". Таким образом, он лениво прекращает анализ, когда встречает первый , несмотря на то что подобное соответствие может встретиться далее еще не один раз. Хотя со словом *ленивый* (lazy) обычно ассоциируется нечто отрицательное, это поведение — именно то, что нам нужно в данном случае.

Все метасимволы повторения (*, +, ?, {n}, {n,} и {n,m}) можно сопроводить вопросительным знаком, чтобы сделать их ленивыми (хотя на практике я использовал его только для * и +).

Обратные ссылки

Группировка обеспечивает еще одну возможность — *обратные ссылки* (backreference). В моей практике это одно из наименее используемых средств регулярного выражения, но есть один случай, когда без него не обойтись. Прежде чем рассмотреть действительно полезный пример, давайте сначала рассмотрим глупый.

Предположим, что вы хотите распознавать названия поп-групп согласно шаблону XYYX (держу пари, вы можете вспомнить названия реальных поп-групп, которые

удовлетворяют этому шаблону). Таким образом, мы хотим распознать PJJP, GOOG и ABBA. Вот где в игру вступают обратные ссылки. Каждой группе (включающей подгруппы) в регулярном выражении присваивается номер, слева направо, начиная с 1. Вы можете обратиться к этой группе в регулярном выражении, используя обратную косую черту, после которой указан ее номер. Другими словами, `\1` означает "соответствие первой группы". Непонятно? Давайте рассмотрим пример.

```
const promo = "Opening for XAAX is the dynamic GOOG! At the box office now!";
const bands = promo.match(/(?:[A-Z])(?:[A-Z])\2\1/g);
```

Читая слева направо, мы видим, что есть две группы, а затем `\2\1`. Так, если первая группа соответствует `X` и вторая группа соответствует `A`, то `\2` должно соответствовать `A` и `\1` должно соответствовать `X`.

Если это кажется вам замечательным, но не очень полезным, вы не одиноки. Единственный случай, когда я полагаю, что обратные ссылки полезны (кроме случаев решения головоломок), — это распознавание кавычек.

В HTML вы можете использовать одиночные или двойные кавычки для указания значений атрибутов. Это можно сделать так.

```
// здесь мы используем обратные апострофы, поскольку одиночные
// и двойные кавычки применяются в коде:
const html = '<img alt='A "simple" example.'>' +
             '<img alt="Don't abuse it!">';
const matches = html.match(/<img alt=(?:['"]).*?\1/g);
```

Обратите внимание, что в этом примере есть некоторое упрощение: если атрибут `alt` не будет указан первым, наше регулярное выражение не сработает. Точно также оно не сработает, если перед `alt` будет указан дополнительный пробел. Позже мы вернемся к этому примеру и решим проблему.

Как и раньше, первая группа будет распознавать одиночную или двойную кавычку, сопровождаемую любым количеством символов (обратите внимание на вопросительный знак, который делает распознавание ленивым), за которыми следует `\1` — т.е. любое первое соответствие одинарной или двойной кавычки.

Давайте закрепим понятие ленивых и жадных соответствий. Удалим вопросительный знак после `*`, сделав распознавание жадным. Запустите выражение снова. Что вы видите? Понимаете, почему так получилось? Это очень важная концепция для понимания регулярных выражений, поэтому, если остались неясности, я рекомендую вам прочитать еще раз раздел по ленивому и жадному распознаваниям.

Группы замены

Одним из преимуществ, предоставляемых группировкой, является способность выполнять более сложные замены. Продолжая наш пример с HTML-кодом, скажем,

что мы хотим отбросить все лишнее из дескриптора <a> и оставить только его URL, указанный в атрибуте href.

```
let html = '<a class="nope" href="/yep">Yep</a>';
html = html.replace(/<a .*?(href=".*?").*?>/, '<a $1>');
```

Подобно обратным ссылкам, всем группам присваиваются номера начиная с 1. В самом регулярном выражении мы обращаемся к первой группе как к \1; в строке для замены мы должны использовать $1. Обратите внимание на использование в этом регулярном выражении ленивых квалификаторов, чтобы оно не распространялось на несколько дескрипторов <a>. Это регулярное выражение не сработает, если в атрибуте href будут использоваться одинарные кавычки вместо двойных.

Давайте дополним пример. Сохраним только атрибуты class и href и ничего более.

```
let html = '<a class="yep" href="/yep" id="nope">Yep</a>';
html = html.replace(/<a .*?(class=".*?").*?(href=".*?").*?>/, '<a $2 $1>');
```

Обратите внимание: в этом регулярном выражении мы изменяем порядок следования атрибутов class и href на обратный, чтобы первым шел атрибут href. Проблема этого регулярного выражения в том, что если атрибуты class и href не располагаются друг за другом (как уже упоминалось выше), а также, если в них будут использоваться одинарные кавычки вместо парных, оно не сработает. В следующем разделе мы увидим еще более сложное решение.

Кроме выражений $1, $2, существуют также $' (все перед соответствием), $& (само соответствие) и $' (все после соответствия). Если вы хотите использовать литеральный знак доллара, используйте $$.

```
const input = "One two three";
input.replace(/two/, '($')');  // "One (One ) three"
input.replace(/\w+/g, '($&)'); // "(One) (two) (three)"
input.replace(/two/, "($')");  // "One ( three) three"
input.replace(/two/, "($$)");  // "One ($) three"
```

Этими макросами замены часто пренебрегают, но я видел их использование в очень хитрых решениях, так что не забывайте о них!

Функции замены

Это мое любимое средство регулярных выражений, которое зачастую позволяет разделять очень сложные регулярные выражения на несколько более простых.

Давайте снова рассмотрим практический пример изменения элементов HTML-кода. Предположим, что вы пишете программу, которая преобразовывает все ссылки <a> в очень специфический формат: вы хотите сохранить атрибуты class, id и href, но удалить все остальные. Проблема в том, что код на входе может быть беспорядочным. Атрибуты присутствуют не всегда, а когда они есть, вы не можете

гарантировать, что они будут следовать в том же порядке. Таким образом, нужно учитывать следующие варианты исходного кода (среди многих других).

```
const html =
    '<a class="foo" href="/foo" id="foo">Foo</a>\n' +
    '<A href='/foo' Class="foo">Foo</a>\n' +
    '<a href="/foo">Foo</a>\n' +
    '<a onclick="javascript:alert('foo!')" href="/foo">Foo</a>';
```

К настоящему времени вам необходимо понимать, что это трудная задача для регулярного выражения: слишком много возможных вариантов! Однако мы можем значительно сократить количество вариантов, разделив это регулярное выражение на *два*: одно — для распознавания дескрипторов `<a>` и второе — для замены содержимого дескрипторов `<a>` только тем, что вы хотите.

Давайте сначала рассмотрим вторую задачу. Если все, чего вы хотели, — это только дескриптор `<a>`, а все атрибуты, кроме `class`, `id` и `href`, можно отбросить, то задача куда проще. Но даже в этом случае, как мы видели ранее, может возникнуть проблема, если мы не сможем гарантировать, что атрибуты следуют в определенном порядке. Есть несколько способов решить эту задачу, но мы будем использовать `String.prototype.split`, чтобы просматривать атрибуты по одному.

```
function sanitizeATag(aTag) {
    // получить части дескриптора...
    const parts = aTag.match(/<a\s+(.*?)>(.*?)<\/a>/i);
    // parts[1] - атрибуты открывающего дескриптора <a>
    // parts[2] - то, что между дескрипторами <a> и </a>
    const attributes = parts[1]
        // теперь разделяем на отдельные атрибуты
        .split(/\s+/);
    return '<a ' + attributes
        // мы хотим только атрибуты class, id и href
        .filter(attr => /^(?:class|id|href)[\s=]/i.test(attr))
        // соединить через пробел
        .join(' ')
        // закрыть открытый дескриптор <a>
        + '>'
        // добавить содержимое
        + parts[2]
        // и завершающий дескриптор
        + '</a>';
}
```

Эта функция длиннее, чем могла бы быть, но мы разредили ее для ясности. Обратите внимание, что даже в этой функции мы используем несколько регулярных выражений: одно — чтобы распознать части дескриптора `<a>`, одно — для разделения (регулярное выражение используется для идентификации одного или нескольких

пробельных символов) и одно — для фильтрации только желательных атрибутов. Было бы куда труднее сделать все это в одном регулярном выражении.

Теперь интересная часть: использование функции `sanitizeATag` в блоке HTML, который, кроме прочего HTML-кода, может содержать много дескрипторов `<a>`. Достаточно просто написать регулярное выражение для распознавания только дескрипторов `<a>`.

```
html.match(/<a .*?>(.*?)<\/a>/ig);
```

Но что нам с этим делать? Как можно передать *функцию* в `String.prototype.replace` в качестве параметра замены. До сих пор в качестве параметра замены мы использовали только строки. Функция позволяет предпринять специальное действие для *каждой замены*. Прежде чем закончить свой пример, давайте используем `console.log`, чтобы увидеть, как это работает.

```
html.replace(/<a .*?>(.*?)<\/a>/ig, function(m, g1, offset) {
    console.log('Дескриптор <a> найден в позиции ${offset}. Содержимое:
${g1}');
});
```

Функция, которую вы передаете в `String.prototype.replace`, получает следующие аргументы по порядку.

- Вся соответствующая строка (эквивалент `$&`).
- Соответствующая группа (если есть). Таких аргументов будет столько, сколько есть групп.
- Смещение в пределах исходной строки (число), где произошло распознавание.
- Исходная строка (используется редко).

Здесь возвращаемое значение функции используется для замены текста в возвращенной строке. В данном примере мы только регистрируем факт на консоли, но ничего не возвращаем из функции. Таким образом, из функции будет возвращено значение `undefined`, которое затем преобразовывается в строку и используется для замены. Задачей этого примера была механика, а не фактическая замена; здесь мы просто отбрасываем получающуюся строку.

Теперь вернемся к нашему примеру. У нас уже есть своя функция для санации дескриптора `<a>` и способ для поиска дескрипторов `<a>` в блоке HTML, поэтому мы можем просто их совместить.

```
html.replace(/<a .*?<\/a>/ig, function(m) {
    return sanitizeATag(m);
});
```

Мы можем упростить это еще больше — полагая, что параметры в функции `sanitizeATag` точно соответствуют тому, что передает `String.prototype.replace`,

можно избавиться от анонимной функции и использовать `sanitizeATag` непосредственно.

```
html.replace(/<a .*?<\/a>/ig, sanitizeATag);
```

Надеемся, что мощь этих функциональных возможностей очевидна. Всякий раз, сталкиваясь с задачей, подразумевающей распознавание малых строк в пределах большей строки и обработку меньших строк, помните, что можно передать функцию в `String.prototype.replace`!

Привязка

Очень часто приходится учитывать вещи, которые должны происходить в начале или конце строки, а также во всей строке сразу, а не в ее части. Здесь нам пригодятся *якоря* (anchor). Есть два якоря: `^`, который соответствует началу строки, и `$`, который соответствует концу строки.

```
const input = "It was the best of times, it was the worst of times";
const beginning = input.match(/^\w+/g);   // "It"
const end = input.match(/\w+$/g);         // "times"
const everything = input.match(/^.*$/g);  // то же, что и на входе
const nomatch1 = input.match(/^best/ig);
const nomatch2 = input.match(/worst$/ig);
```

У якорей есть еще один нюанс, о котором следует знать. Обычно они соответствуют началу и концу *всей строки*, даже если в ней есть символы новой строки. Если вы хотите обработать строку как многострочную (разделенную символами новой строки), используйте параметр `m` (multiline).

```
const input = "One line\nTwo lines\nThree lines\nFour";
const beginnings = input.match(/^\w+/mg); // ["One", "Two", "Three", "Four"]
const endings = input.match(/\w+$/mg);    // ["line", "lines", "lines", "Four"]
```

Распознавание границ слов

Одним из малоиспользуемых, но полезных элементов регулярных выражений, являются *границы слова*. Подобно якорям начала и конца строки, метасимвол границы слова `\b` и его инверсия `\B` *не перерабатывает входные данные*. Это может быть очень удобным свойством, как мы вскоре убедимся.

Граница слова определяется местом, где метасимволу `\w` предшествует или следует после него метасимвол `\W` (не слово) либо символ начала или конца строки. Предположим, что вы пытаетесь заменить адреса электронной почты в английском тексте гиперссылками (в этом примере мы подразумеваем, что адреса электронной почты начинаются с символа и заканчиваются символом). Обсудим ситуации, которые стоит рассмотреть.

```
const inputs = [
    "john@doe.com",                 // только адрес
    "john@doe.com is my email",     // адрес вначале
    "my email is john@doe.com",     // адрес в конце
    "use john@doe.com, my email", // адрес в середине с запятой после
    "my email:john@doe.com.",       // адрес окружен пунктуацией
];
```

Учесть следует много, но все эти адреса электронной почты существуют в границах слова. Еще одно преимущество маркеров границ слов в том, что, поскольку они не перерабатывают входные данные, мы не должны заботиться об их "откладывании" в строке замены.

```
const emailMatcher =
    /\b[a-z][a-z0-9._-]*@[a-z][a-z0-9_-]+\.[a-z]+(?:\.[a-z]+)?\b/ig;
inputs.map(s => s.replace(emailMatcher, '<a href="mailto:$&">$&</a>'));
// возвращает [
//    "<a href="mailto:john@doe.com">john@doe.com</a>",
//    "<a href="mailto:john@doe.com">john@doe.com</a> is my email",
//    "my email is <a href="mailto:john@doe.com">john@doe.com</a>",
//    "use <a href="mailto:john@doe.com">john@doe.com</a>, my email",
//    "my email:<a href="mailto:john@doe.com>john@doe.com</a>.",
// ]
```

Кроме маркеров границ слов, в этом регулярном выражении используется много средств, которые мы рассматривали в данной главе. На первый взгляд, оно кажется просто обескураживающим, выполнив его анализ, можно сделать хороший шаг на пути к мастерству овладения регулярными выражениями (обратите особое внимание на то, что макрос замены, `$&`, *не* включает символы, окружающие адрес электронной почты... поскольку они не были переработаны).

Границы слов также удобны тогда, когда вы пытаетесь искать текст, который начинается с или заканчивается с определенного слова или содержит другое слово. Например, `/\bcount/` найдет слова *count* и *countdown*, но не *discount*, *recount* и *accountable*; `/\bcount\B/` найдет только *countdown*; `/\Bcount\b/` найдет *discount* и *recount*, а `/\Bcount\B/` найдет только *accountable*.

Упреждения

Если использование жадного и ленивого распознавания отделяют дилетантов от профессионалов, то *упреждения* (lookahead) отделяют профессионалов от гуру. Упреждения (подобно якорям и метасимволам границ слова) не перерабатывают входные данные. Однако в отличие от якорей и границ слова, они универсальны: вы можете распознать любое подвыражение, не перерабатывая его. Как и с метасимволами границы слова, тот факт, что упреждения не распознаются (match), может предохранить вас от необходимости "откладывать" нечто при выполнении замены.

Хотя это и хороший пример, здесь упреждения можно и не использовать. Упреждения необходимы, когда содержимое перекрывается; они могут также упростить определенные типы поиска.

Классический пример — проверка соблюдения некой политики при выборе паролей. Для простоты предположим, что наш пароль состоит только из букв и цифр, и что он должен содержать по крайней мере одну прописную букву, число и строчную букву. Конечно, мы могли бы использовать несколько регулярных выражений.

```
function validPassword(p) {
    return /[A-Z]/.test(p) &&     // по крайней мере одна прописная буква
        /[0-9]/.test(p) &&        // по крайней мере одна цифра
        /[a-z]/.test(p) &&        // по крайней мере одна строчная буква
        !/[^a-zA-Z0-9]/.test(p); // только буквы и цифры
}
```

Давайте объединим это в одно регулярное выражение. Наша первая попытка провалится.

```
function validPassword(p) {
    return /[A-Z].*[0-9][a-z]/.test(p);
}
```

Это выражение не только требует, чтобы прописная буква располагалась перед цифрой и строчной буквой, но и не позволяет распознать запрещенные символы вообще. И действительно нет никакого разумного способа сделать это хотя бы потому, что символы будут переработаны в процессе обработки регулярного выражения.

Здесь нам на помощь приходят упреждения. По существу, каждое упреждение — это независимое регулярное выражение, которое не перерабатывает входные данные. Упреждения в JavaScript выглядят как `(?=<подвыражение>)`. Есть также "негативное упреждение" (negative lookahead): `(?!<подвыражение>)`, оно будет распознавать только то, что *не соответствует* подвыражению. Теперь мы можем написать единое регулярное выражение для проверки наших паролей.

```
function validPassword(p) {
    return /(?=.*[A-Z])(?=.*[0-9])(?=.*[a-z])(?!.*[^a-zA-Z0-9])/.test(p);
}
```

Взглянув на эту абракадабру вы можете подумать, что та функция из нескольких регулярных выражений лучше или по крайней мере понятнее. В этом примере я, вероятно, согласился бы с вами. Но в нем показан один из важнейших случаев использования упреждений (и негативных упреждений). Упреждения, определенно, относятся к категории "высшего пилотажа" в регулярных выражениях, но они важны для решения определенных задач.

Динамическое создание регулярных выражений

В начале этой главы упоминалось, что литеральный синтаксис регулярных выражений предпочтительнее конструктора `RegExp`. Кроме необходимости вводить вчетверо меньше символов, литералы регулярных выражений предпочтительнее потому, что в них не нужно экранировать символы обратной косой черты, как это делается в строках JavaScript. На самом деле конструктор `RegExp` необходимо использовать, чтобы создавать регулярные выражения *динамически*. Например, у вас мог бы быть массив имен пользователей, которые вы хотите искать в строке; нет никакого разумного способа ввести все эти имена пользователей в литерал регулярного выражения. Вот где пригодится конструктор `RegExp`, поскольку он создает регулярное выражение из строки, которая *может* быть построена динамически. Давайте рассмотрим пример.

```
const users = ["mary", "nick", "arthur", "sam", "yvette"];
const text = "User @arthur started the backup and 15:15, " +
    "and @nick and @yvette restored it at 18:35.";
const userRegex = new RegExp('@(?:${users.join('|')})\\b', 'g');
text.match(userRegex); // [ "@arthur", "@nick", "@yvette" ]
```

Эквивалентное литеральное регулярное выражение для этого примера было бы таким: `/@(?:mary|nick|arthur|sam|yvette)\b/g`, но мы сумели создать его динамически. Обратите внимание на использование двух обратных косых черточек перед `b` (метасимвол границы слова); первая косая черта экранирует вторую косую черту в строке.

Заключение

Хотя в данной главе затронуты основные моменты регулярных выражений, это сделано только поверхностно. Чтобы стать профессионалом в регулярных выражениях нужно приблизительно 20% времени уделить теории и 80% практике. Пока вы новичок (и даже когда вы наберетесь опыта!) при изучении материала вам поможет надежный тестер регулярных выражений (такой, как сайт `https://regex101.com/#javascript`). Самое важное, что вы должны были уяснить из этой главы, — понимание того, как обработчик регулярного выражения перерабатывает входные данные; недостаточное понимание в этой области является причиной большинства проблем.

JavaScript в браузере

JavaScript был задуман как язык сценариев браузера и сегодня удерживает почти полную монополию в этой роли. Данная глава предназначена для тех, кто работает с JavaScript в браузере. Для браузера язык JavaScript практически не меняется, но при его использовании в этой среде сделан ряд ограничений и реализован специальный интерфейс API.

Полное описание процесса создания приложений на JavaScript для браузера достойно отдельной книги. Цель этой главы — ознакомить читателя с основными концепциями разработки приложений для браузера, чтобы обеспечить глубокое понимание основ. В конце этой главы рекомендовано несколько дополнительных учебных пособий.

ES5 или ES6?

Полагаю, что вы уже убедились в удобстве дополнений, предоставляемых ES6. К сожалению, должно пройти еще некоторое время, прежде чем вы сможете полагаться на полную и единообразную поддержку ES6 на веб-страницах.

На стороне сервера вы можете знать наверняка, какие средства ES6 поддерживаются (при наличии контроля над интерпретатором JavaScript). На веб-странице вы пересылаете свой драгоценный код по протоколу HTTP(S) браузеру, где он выполняется неким интерпретатором JavaScript, который вы не контролируете. Хуже того, у вас может даже не быть надежной информации об используемом типе браузера.

Эту проблему решают так называемые “вечнозеленые” браузеры; при автоматическом обновлении (без запроса к пользователю) они позволяют быстро и последовательно устанавливать более новые веб-стандарты. Но это решает проблему только частично, не устраняя ее.

Если вы не можете так или иначе контролировать среду своего пользователя, в обозримом будущем вам придется публиковать код ES5. Это не конец света: транскомпиляция предоставляет вполне доступный путь для написания кода ES6 сегодня. Она может существенно облегчить развертывание и отладку, но такова цена прогресса.

В этой главе мы подразумеваем использование транскомпилятора, как было описано в главе 2. Все примеры этой главы правильно выполняются в последней версии

браузера Firefox без транскомпиляции. Если вы будете публиковать свой код для более широкой аудитории, то его придется транскомпилировать, чтобы гарантировать его надежную работу на многих других браузерах.

Объектная модель документа

Объектная модель документа (Document Object Model — DOM) — это соглашение для описания структуры HTML-документа, находящееся в основе взаимодействия с браузером.

Концептуально DOM — это дерево. Дерево состоит из *узлов* (node): у каждого узла есть родитель (за исключением корневого узла) и любое количество *дочерних узлов* (child node). Корневой узел — это *документ* (document); он имеет один дочерний узел, которым является элемент `<html>`. У элемента `<html>`, в свою очередь, есть два дочерних узла: элементы `<head>` и `<body>`. Пример DOM приведен на рис. 18.1.

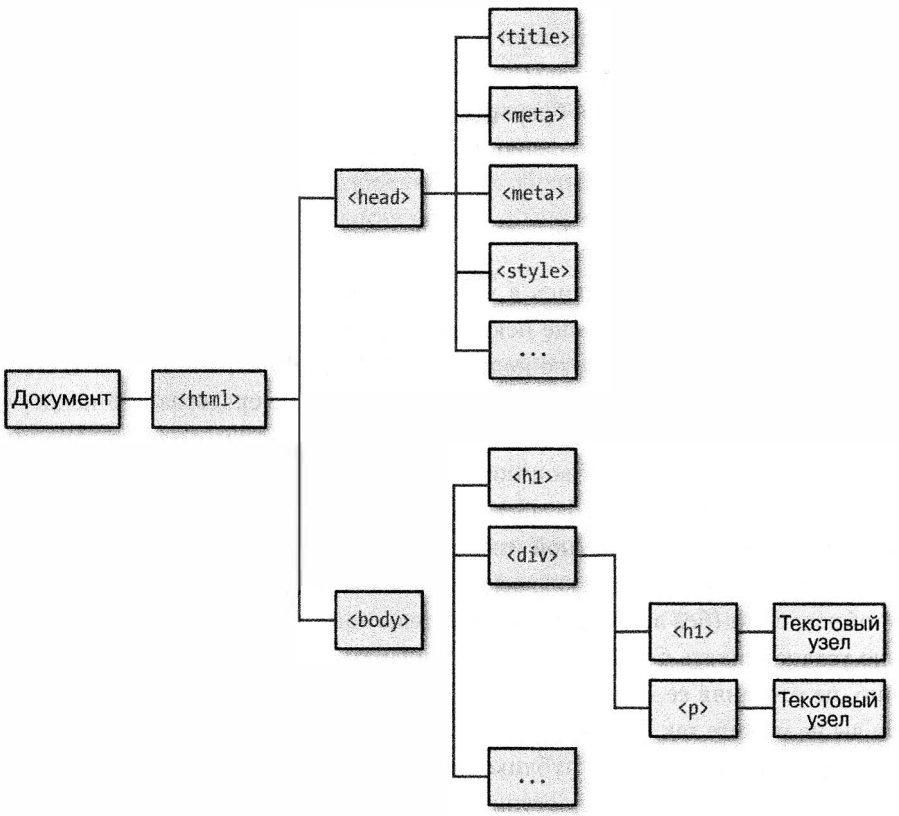


Рис. 18.1. Дерево DOM

Каждый узел в дереве DOM (включая сам документ) является экземпляром класса `Node` (не путать с `Node.js`, это тема следующей главы). У объектов `Node` есть свойства `parentNode` и `childNodes`, а также такие свойства идентификации, как `nodeName` и `nodeType`.

Модель DOM полностью состоит из узлов, но только *некоторые из них* являются *элементами* (element) HTML. Например, дескриптор абзаца (`<p>`) *является HTML-элементом*, но текст, который он содержит, является *текстовым узлом* (text node). Очень часто термины *узел* и *элемент* используются как синонимы, что редко вводит в заблуждение, но технически неправильно. В этой главе мы будем (по большей части) иметь дело с узлами, которые являются HTML-элементами, и когда мы говорим "элемент", мы подразумеваем "узел элемента".

Для демонстрации возможностей в следующих примерах мы будем использовать очень простой HTML-файл. Создайте следующий файл по имени `simple.html`.

```
<!doctype html>
<html>
    <head>
        <meta charset="utf-8">
        <title>Простая HTML-страница</title>
        <style>
            .callout {
                border: solid 1px #ff0080;
                margin: 2px 4px;
                padding: 2px 6px;
            }
            .code {
                background: #ccc;
                margin: 1px 2px;
                padding: 1px 4px;
                font-family: monospace;
            }
        </style>
    </head>
    <body>
        <header>
            <h1> Простая HTML-страница</h1>
        </header>
        <div id="content">
            <p>Это <i>простой</i> HTML-файл.</p>
            <div class="callout">
                <p>Он позволяет творить удивительные вещи!</p>
            </div>
            <p>Идентификаторы элементов (например, <span
```

```
class="code">#content</span>)
                являются уникальными (они должны использоваться на странице
                только один раз).</p>
            <p>Имена классов (например, <span class="code">.callout</span>)
                могут использоваться во многих элементах.</p>
            <div id="callout2" class="callout fancy">
                <p>Одному HTML-элементу можно присвоить несколько классов.</p>
            </div>
        </div>
    </body>
</html>
```

У каждого узла есть свойства `nodeType` и `nodeName` (среди других). Свойство `nodeType` — это целочисленный идентификатор типа узла. Объект `Node` содержит константы, которые соответствуют этим числам. В этой главе мы прежде всего будем иметь дело с узлами типов `Node.ELEMENT_NODE` (HTML-элементы) и `Node.TEXT_NODE` (текстовое содержимое, которое обычно находится в HTML-элементах). Более подробная информация по этой теме приведена в *части `nodeType` документации MDN* (`https://developer.mozilla.org/ru/docs/Web/API/Node/nodeType`).

Ниже приведено хорошее упражнение по написанию функции, которая *обходит* (traverse) все дерево DOM и выводит его на консоль, начиная с `document`.

```
function printDOM(node, prefix) {
    console.log(prefix + node.nodeName);
    for(let i=0; i<node.childNodes.length; i++) {
        printDOM(node.childNodes[i], prefix + '\t');
    }
}
printDOM(document, '');
```

Эта рекурсивная функция осуществляет *обход дерева в глубину в прямом порядке* (depth-first, pre-order traversal). Таким образом, она следует вниз по всей ветви, прежде чем перейти к следующей ветви. Если вы запустите этот пример в браузере с загруженной страницей, то увидите на консоли всю структуру страницы.

Хотя это упражнение поучительно, такой способ манипулирования HTML-кодом (подразумевающий необходимость обойти все дерево DOM для нахождения искомого узла) был бы утомителен и неэффективен. К счастью, DOM предоставляет более эффективные методы поиска HTML-элементов.

Хотя написание собственной функции обхода дерева является хорошим упражнением, API DOM предоставляют объект `TreeWalker`, который позволяет перебрать все элементы DOM (с опциональной фильтрацией по элементам определенных типов). Более подробная информация по этой теме приведена в части `document.createTreeWalker` документации MDN.

Немного терминологии

Концепция дерева проста и интуитивно понятна, что ведет к интуитивно понятной терминологии. Родитель узла — это его *прямой* (direct) предок (т.е. не "дедушка"), а дочерний узел — это *прямой* потомок (т.е. не "внук"). Термин *потомок* (descendant) используется для описания дочернего или дочернего для дочернего узла или еще более дальнего. Термин *предок* (ancestor) используется для описания родителя, родителя родителя и т.д.

Методы-получатели модели DOM

Модель DOM предоставляет методы-получатели (`get...`), которые позволяют быстро находить определенные элементы HTML.

Первый из них — это метод `document.getElementById`. Каждому HTML-элементу на веб-странице может быть присвоен уникальный идентификатор, а метод `document.getElementById` позволяет получить этот элемент по его идентификатору.

```
document.getElementById('content'); // <div id="content">...</div>
```

Браузеры не делают ничего для проверки уникальности идентификаторов (хотя HTML-валидатор эти проблемы учитывает), поэтому соблюдение уникальности идентификаторов возложено исключительно на вас. Поскольку создание веб-страниц (с компонентами, получаемыми из разных источников) становится все более и более сложной задачей, все более и более усложняется задача избежания повторов идентификаторов. Поэтому я рекомендую использовать идентификаторы экономно и очень внимательно.

Метод `document.getElementsByClassName` возвращает коллекцию элементов, у которых есть заданное имя класса.

```
const callouts = document.getElementsByClassName('callout');
```

Метод `document.getElementsByTagName` возвращает коллекцию элементов, у которых есть заданное имя дескриптора.

```
const paragraphs = document.getElementsByTagName('p');
```

Все методы DOM, которые возвращают коллекцию, фактически возвращают не массив JavaScript, а экземпляр объекта `HTMLCollection`, который "подобен массиву". Вы можете перебрать его в цикле `for`, но методы `Array.prototype` (такие, как `map`, `filter` и `reduce`), не будут доступны. Вы можете преобразовать объект `HTMLCollection` в массив, используя оператор расширения `[...document.getElementsByTagName(p)]`.

Выборка элементов DOM

Методы `getElementById`, `getElementsByClassName` и `getElementsByTagName` полезны, но есть намного более общий (и мощный) метод, который может находить элементы не только по одному условию (идентификатор, класс или имя дескриптора), но и по *отношению* элемента *к другим элементам*. Для этой цели в методах объекта `document` `querySelector` и `querySelectorAll` можно использовать *селекторы CSS* (CSS selector).

Селекторы CSS позволяют идентифицировать элементы по именам (`<p>`, `<div>` и т.д.), идентификатору, классу (или комбинации классов) либо по любой их комбинации. Чтобы идентифицировать элементы по имени, достаточно использовать название элемента (без угловых скобок). Таким образом, `a` будет соответствовать всем дескрипторам `<a>` в DOM, а `br` — всем дескрипторам `<br>`. Чтобы идентифицировать элементы по их классу, используйте точку перед именем класса: `.callout` будет соответствовать всем элементам класса `callout`. Чтобы распознать несколько классов, достаточно разделить их точками: `.callout.fancy` будут соответствовать все элементы класса `callout` *и* класса `fancy`. Наконец они могут быть объединены; например, `a#callout2.callout.fancy` будет соответствовать элементам `<a>` с идентификатором `callout2` и классами `callout` и `fancy` (очень редко встретишь селектор, в котором одновременно используется имя элемента, идентификатор и класс (классы)... но это возможно!).

Наилучший способ приобретения навыков в селекторах CSS — это загрузить в браузер пример HTML, предоставленный в данной главе, открыть консоль браузера и опробовать их с методом `querySelectorAll`. Введите, например, на консоли **`document.querySelectorAll('.callout')`**. Все примеры из этого раздела подобраны так, чтобы в результате вызова метода `querySelectorAll` на консоль был выведен хотя бы один результат.

До сих пор мы говорили об идентификации определенных элементов независимо от того, где они присутствуют в DOM. Селекторы CSS также позволяют находить элементы согласно их позиции в DOM.

Если разделить несколько элементов селекторов пробелами, можно выбрать узлы с определенной иерархией. Например, `#content p` выберет элементы `<p>`, которые являются *потомками* любого элемента, имеющего идентификатор `content`. Аналогично `#content div p` выберет элементы `<p>`, являющиеся потомками элемента `<div>`, который является потомком элемента с идентификатором `content`.

Если разделить несколько элементов селектора знаком "больше" (`>`), можно выбрать узлы, которые являются *прямыми потомками* указанного элемента. Например, `#content > p` выберет элементы `<p>`, которые находятся в элементе с идентификатором `content` (сравните это с `"#content p"`).

Обратите внимание, что вы можете объединить предков с прямыми потомками. Например, `body .content > p` выберет дескрипторы <p>, которые являются прямыми потомками элементов класса `content`, которые являются потомками дескриптора <body>.

Существуют куда более сложные селекторы, но здесь рассматриваются лишь наиболее распространенные. Более подробная информация по этой теме приведена в части по селекторам в документации MDN (`https://developer.mozilla.org/ru/docs/Learn/CSS/Introduction_to_CSS/Selectors`).

Манипулирование элементами DOM

Теперь, когда известно, как обходить, находить и выбирать элементы, возникает вопрос "Что с ними *делать*?" Начнем с модификации содержимого. У каждого элемента есть два свойства, `textContent` и `innerHTML`, которые позволяют получить доступ к содержимому элемента и изменить его. Свойство `textContent` содержит только "голые" текстовые данные HTML-дескриптора, а `innerHTML` позволяет использовать HTML-код (что приводит к образованию новых узлов DOM). Давайте рассмотрим, как можно обратиться к первому абзацу в нашем примере и изменить его.

```
const para1 = document.getElementsByTagName('p')[0];
para1.textContent; // "Это простой HTML-файл."
para1.innerHTML;   // "Это <i>простой</i> HTML-файл."
para1.textContent = "Измененный HTML-файл"; // Посмотрите в браузере,
para1.innerHTML = "<i>Измененный</i> HTML-файл";    // что изменилось!
```

Присвоение значений свойствам `textContent` и `innerHTML` является *деструктивной* операцией: она заменят то, что находится в элементе, независимо от его размера или сложности. Например, можно заменить сразу все содержимое веб-страницы, изменив свойство `innerHTML` элемента <body>!

Создание новых элементов DOM

Мы уже видели, как можно неявно создать новый узел DOM, изменив значение свойства `innerHTML` элемента. Но можно и явно создать новый узел, используя метод `document.createElement`. Он создает новый элемент, но не добавляет его в дерево DOM; вам нужно будет сделать это самостоятельно позже. Давайте создадим два новых элемента абзаца; один станет первым абзацем в блоке , а второй — последним.

```
const p1 = document.createElement('p');
const p2 = document.createElement('p');
p1.textContent = "Это было создано динамически!";
p2.textContent = "И это тоже было создано динамически!";
```

Чтобы добавить эти вновь созданные элементы в DOM используются методы `insertBefore` и `appendChild`. Но сначала мы должны будем получить ссылки на родительский элемент DOM (`<div id="content">`) и его первый дочерний узел.

```
const parent = document.getElementById('content');
const firstChild = parent.childNodes[0];
```

Теперь можно вставить вновь созданные элементы.

```
parent.insertBefore(p1, firstChild);
parent.appendChild(p2);
```

Методу `insertBefore` передается сам вставляемый элемент и ссылка на узел, перед которым должна быть выполнена вставка. Метод `appendChild` очень прост, он просто добавляет определенный элемент в конец списка узлов данного элемента (т.е. создает последний дочерний узел).

Применение стилей к элементам

API DOM обеспечивают полный контроль над стилями элементов. Однако вместо изменения свойств индивидуальных элементов хорошей практикой обычно считается использование классов CSS. Таким образом, если вы хотите изменить стиль элемента, создайте новый класс CSS, а затем примените его к элементу (или элементам), стиль которого собираетесь изменить. Используя JavaScript, довольно просто применить существующий класс CSS к элементу. Например, если нам нужно выделить все абзацы, в которых содержится слово *уникальный*, то сначала создадим новый класс CSS.

```
.highlight {
    background: #ff0;
    font-style: italic;
}
```

Теперь можно найти все дескрипторы `<p>` и, если они содержат слово *уникальный*, добавить к ним класс `highlight`. У каждого элемента есть свойство `classList`, которое содержит все имеющиеся у элемента классы (если они есть). У свойства `classList` есть метод `add`, который позволяет добавлять новые классы. Мы будем использовать этот пример далее в главе, поэтому поместим его в функцию по имени `highlightParas`.

```
function highlightParas(containing) {
    if(typeof containing === 'string')
        containing = new RegExp('\\b${containing}\\b', 'i');
    const paras = document.getElementsByTagName('p');
    console.log(paras);
    for(let p of paras) {
        if(!containing.test(p.textContent)) continue;
        p.classList.add('highlight');
```

```
    }
}
highlightParas('уникальный');
```

А впоследствии, если мы захотим удалить выделение, мы сможем использовать метод `classList.remove`.

```
function removeParaHighlights() {
    const paras = document.querySelectorAll('p.highlight');
    for(let p of paras) {
        p.classList.remove('highlight');
    }
}
```

При удалении класса `highlight` можно многократно использовать одну и ту же переменную `paras` и просто вызвать метод `remove('highlight')` для каждого элемента абзаца. Однако это не сработает, если элементу еще не был назначен класс. Но вероятнее всего, удаление нужно будет выполнить в некий более поздний момент времени, когда, возможно, будут выделены абзацы, добавленные другим кодом. Поэтому если наше намерение заключается в том, чтобы снять все, что выделено, использование метода `querySelectorAll` является самым надежным способом.

Атрибуты данных

В HTML5 введены *атрибуты данных* (data attribute), которые позволяют добавлять произвольные данные к HTML-элементам; браузером эти данные не отображаются, но они позволяют добавлять к элементам информацию, легко читаемую и изменяемую с помощью JavaScript. Давайте изменим наш HTML-код, добавив кнопки, которые в конечном счете соединим с нашими функциями `highlightParas` и `removeParaHighlights`.

```
<button data-action="highlight" data-containing="уникальный">
    Выделяет абзацы, содержащие слово "уникальный"
</button>
<button data-action="removeHighlights">
    Удаляет выделение
</button>
```

Назовем наши атрибуты данных `action` и `contains` (имена мы выбираем сами), и используем `document.querySelectorAll` для поиска всех элементов, имеющих действие `"highlight"`.

```
const highlightActions = document.querySelectorAll('[data-
action="highlight"]');
```

Тем самым мы ввели новый тип селектора CSS. До сих пор мы сталкивались с селекторами, которые соответствовали определенным именам дескрипторов, классов и идентификаторов. Синтаксис с использованием квадратных скобок позволяет искать элементы с любым атрибутом... в данном случае — с определенным атрибутом данных.

Поскольку у нас есть только одна кнопка, мы можем использовать метод `querySelector` вместо `querySelectorAll`. Однако последний позволяет нам обрабатывать сразу несколько элементов, предназначенных для выполнения одного и того же действия (что весьма распространено: вспомните о действиях, которые можно выполнить через меню, ссылку или панель инструментов, и все на той же странице). Если обратить внимание на один из элементов в `highlightActions`, то можно заметить, что у него есть свойство `dataset`.

```
highlightActions[0].dataset;
// DOMStringMap { containing: "уникальный", action: "highlight" }
```

Согласно API DOM значения атрибутов данных хранятся в виде строк (как и предполагалось при реализации класса `DOMStringMap`). Таким образом, вы не можете хранить в атрибутах данных объектные данные. jQuery расширяет функциональные возможности атрибутов данных, предоставляя интерфейс, который позволяет хранить объекты как атрибуты данных, о чем мы узнаем в главе 19.

Используя JavaScript, можно также изменять или добавлять атрибуты данных. Например, если бы мы хотели выделить абзацы со словом *жираф* и указать, что регистр символов имеет значение, то мы могли бы поступить так.

```
highlightActions[0].dataset.containing = "жираф";
highlightActions[0].dataset.caseSensitive = "true";
```

События

В API DOM описано почти 200 событий, и в каждом браузере дополнительно реализованы свои нестандартные события, поэтому мы, конечно, не будем обсуждать здесь все события, но рассмотрим то, что о них необходимо знать. Начнем с очень простого для понимания события `click`. Мы будем использовать событие `click` для соединения нашей кнопки "выделения" с нашей функцией `highlightParas`.

```
const highlightActions = document.querySelectorAll('[data-
action="highlight"]');
for(let a of highlightActions) {
    a.addEventListener('click', evt => {
```

```
        evt.preventDefault();
        highlightParas(a.dataset.containing);
    });
}

const removeHighlightActions =
    document.querySelectorAll('[data-action="removeHighlights"]');
for(let a of removeHighlightActions) {
    a.addEventListener('click', evt => {
        evt.preventDefault();
        removeParaHighlights();
    });
}
```

У каждого элемента есть метод `addEventListener`, который позволяет определять функцию, вызываемую, когда происходит указанное событие. Этой функции передается один аргумент — объект типа `Event`. Объект события содержит всю необходимую информацию о событии, которая специфична для данного типа события. Например, событие `click` будет иметь свойства `clientX` и `clientY`, которые указывают координаты, где произошел щелчок кнопкой мыши, а также свойство `target` — элемент, для которого событие `click` было сгенерировано.

Модель событий спроектирована так, чтобы несколько обработчиков могли обрабатывать одно и то же событие. У многих событий есть стандартные обработчики; например, если пользователь щелкает на ссылке `<a>`, то браузер обработает это событие, загрузив нужную страницу. Если вы хотите предотвратить такое поведение, вызовите метод `preventDefault()` для объекта события. В большинстве обработчиков событий, которые вы пишете, нужно вызывать метод `preventDefault()` (если только вы не хотите *добавить* нечто к стандартному обработчику).

Для выделения текстового абзаца в нашем примере вызывается функция `highlightParas`, которой передается значение элемента данных `containing` кнопки: это позволяет оперативно изменять выделяемое слово, просто отредактировав HTML-код!

Перехват и всплытие событий

Поскольку HTML-документ имеет иерархический характер, события могут быть обработаны в нескольких местах. Например, если вы щелкаете на кнопке, то событие могла бы обработать сама кнопка, родитель кнопки, родитель родителя и т.д. Поскольку обработать событие могут несколько элементов, возникает вопрос "В каком порядке элементы получают возможность отреагировать на событие?"

По существу, есть две возможности. Первая — в предке начиная с самого дальнего. Это так называемый *перехват* (capturing) события. В нашем примере кнопки являются дочерними для элемента `<div id="content">`, который, в свою очередь,

является дочерним для <body>. Поэтому у элемента <body> есть возможность перехватывать события, исходящие от кнопок.

Вторая возможность начинается с элемента, где событие произошло, а затем вверх по иерархии, чтобы у всех предков был шанс отреагировать. Это так называемое *всплытие* (bubbling) событий.

Для поддержки обоих возможностей распространение событий в HTML5 начинается с разрешения обработчикам перехватывать события (начиная с самого дальнего предка и вниз к исходному (target) элементу), а затем события всплывают вверх от исходного элемента к самому дальнему предку.

Для обеспечения возможности вызова дополнительных обработчиков любой обработчик может опционально предпринять одно из трех действий. Первое и наиболее распространенное, которое мы уже видели, — это вызов метода `preventDefault`, который *отменяет* событие. Отмененные события продолжают распространяться, но их свойство `defaultPrevented` устанавливается равным `true`. Встроенные в браузер обработчики событий проверяют значение свойства `defaultPrevented` и, если оно истинно, ничего не предпринимают. Обработчики событий, которые вы пишете, могут проигнорировать значение этого свойства (и обычно так и делают). Второй подход подразумевает вызов метода `stopPropagation`, что предотвращает дальнейшее распространение события за текущий элемент. При этом все обработчики, связанные с текущим элементом, будут вызваны, но никакие из обработчиков, связанные с другими элементами, не вызываются. И наконец (крупный калибр) вызов метода `stopImmediatePropagation` запретит вызов дальнейших обработчиков (даже если они связаны с текущим элементом.

Для демонстрации всех этих действий рассмотрим следующий HTML-код.

```
<!doctype html>
<html>
    <head>
        <title>Распространение событий</title>
        <meta charset="utf-8">
    </head>
    <body>
        <div>
            <button>Щелкни здесь!</button>
        </div>
        <script>

            // это создаст обработчик событий и возвратит его
            function logEvent(handlerName, type, cancel,
                    stop, stopImmediate) {
                // это фактический обработчик событий
                return function(evt) {
                    if(cancel) evt.preventDefault();
                    if(stop) evt.stopPropagation();
```

```
                if(stopImmediate) evt.stopImmediatePropagation();
                console.log('${type}: ${handlerName}' +
                    (evt.defaultPrevented ? ' (отменено)' : ''));
            }
        }

        // это добавляет регистратор события к элементу
        function addEventLogger(elt, type, action) {
            const capture = type === 'capture';
            elt.addEventListener('click',
                logEvent(elt.tagName, type, action==='cancel',
                action==='stop', action==='stop!'), capture);
        }

        const body = document.querySelector('body');
        const div = document.querySelector('div');
        const button = document.querySelector('button');

        addEventLogger(body, 'capture');
        addEventLogger(body, 'bubble');
        addEventLogger(div, 'capture');
        addEventLogger(div, 'bubble');
        addEventLogger(button, 'capture');
        addEventLogger(button, 'bubble');

    </script>
  </body>
</html>
```

Щелкнув на кнопке, вы увидите на консоли следующее.

```
capture: BODY
capture: DIV
capture: BUTTON
bubble: BUTTON
bubble: DIV
bubble: BODY
```

Здесь мы ясно видим процесс перехвата событий, сопровождаемый их всплытием. Обратите внимание, что обработчики элемента, на котором фактически произошло событие, будут вызваны в порядке их добавления, будь то перехват или распространение события (если мы изменим на обратный порядок, в котором мы добавляли обработчики перехвата и всплытия событий, мы увидим, что всплытие следует перед перехватом).

Рассмотрим, что произойдет при отмене распространения события. Изменим пример так, чтобы отменить распространение при перехвате события в `<div>`.

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
```

```
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture');
addEventLogger(button, 'bubble');
```

Здесь распространение продолжается, но событие отмечено как отмененное.

```
capture: BODY
capture: DIV (отменено)
capture: BUTTON (отменено)
bubble: BUTTON (отменено)
bubble: DIV (отменено)
bubble: BODY (отменено)
```

Остановим распространение при перехвате в элементе <button>.

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture', 'stop');
addEventLogger(button, 'bubble');
```

Мы видим, что распространение события остановилось после элемента <button>. Для элемента <button> события все еще генерируются, даже несмотря на то, что произошел перехват события и остановлено его распространение. Элементы и <body>, однако, не получают всплывающие события.

```
capture: BODY
capture: DIV (canceled)
capture: BUTTON (canceled)
bubble: BUTTON (canceled)
```

Наконец выполним немедленную остановку распространения при перехвате события в элементе <button>.

```
addEventLogger(body, 'capture');
addEventLogger(body, 'bubble');
addEventLogger(div, 'capture', 'cancel');
addEventLogger(div, 'bubble');
addEventLogger(button, 'capture', 'stop!');
addEventLogger(button, 'bubble');
```

Распространение события полностью останавливается при перехвате в элементе <button>, никакого дальнейшего распространения не происходит.

```
capture: BODY
capture: DIV (canceled)
capture: BUTTON (canceled)
```

Метод `addEventListener` заменяет уже устаревший способ добавления событий с использованием свойств "`on...`". Например, обработчик щелчка мог бы быть добавлен к элементу `elt` с помощью кода `elt.onclick = function(evt) { /* обработчик */ }`. Основной недостаток этого метода в том, что за раз может быть зарегистрирован только один обработчик.

Хотя маловероятно, что вам придется создавать сложную систему управления распространением событий, очень часто, эта тема вызывает большие затруднения среди новичков. Хорошо понимая детали распространения события, вы сможете возвыситься над толпой.

В библиотеке jQuery обработчики событий могут явно возвращать значение `false`. Это эквивалентно вызову метода `stopPropagation` в обработчике. Данное соглашение jQuery, и оно не распространяется на API DOM.

Категории событий

В MDN есть превосходный справочник по всем *событиям DOM, сгруппированным по категориям*. Вот некоторые из наиболее распространенных категорий событий.

События перетаскивания

Позволяют реализовать интерфейс перетаскивания с использованием таких событий, как `dragstart`, `drag`, `dragend`, `drop` и др.

События фокуса

Позволяют принимать меры, когда пользователь взаимодействует с редактируемыми элементами (такими, как поля формы). Событие `focus` происходит, когда пользователь "переходит" к полю (щелкнув на нем или использовав клавишу <Tab> либо касание), а `blur` — когда пользователь "покидает" поле (щелкнув где-то еще, нажав <Tab> или выполнив касание в другом месте). Событие `change` происходит, когда пользователь вносит изменение в поле.

События формы

Когда пользователь передает данные формы на сервер (щелкнув на кнопке Отправить или нажав клавишу <Enter> в соответствующем контексте), в ней происходит событие `submit`.

События устройства ввода данных

Мы уже встречали событие `click`, но есть и дополнительные события для мыши (`mousedown`, `move`, `mouseup`, `mouseenter`, `mouseleave`, `mouseover`, `mousewheel`)

и клавиатуры (keydown, keypress, keyup). Обратите внимание, что события "касания" (для устройств с сенсорным экраном) имеют приоритет перед событиями мыши, но если сенсорные события не обрабатываются, они приводят к событиям мыши. Например, если пользователь касается кнопки и это событие не будет обработано явно, то будет передано событие click.

События мультимедийной среды

Позволяет отслеживать взаимодействие пользователя с видео и аудиоустройствами в HTML5 (pause, play и т.д.).

События хода выполнения

Сообщают о ходе выполнения работы браузером при загрузке содержимого. Наиболее распространено событие load, происходящее, как только браузер загружает элемент и все его зависимые ресурсы. Событие error также полезно; оно позволяет принять меры, когда элемент недоступен (например, некорректная ссылка на изображение).

События касания

События касания обеспечивают всестороннюю поддержку обработчиков для сенсорных устройств. Разрешается несколько одновременных касаний (посмотрите описание свойства touches элемента события), обеспечивающих сложную сенсорную обработку, такую как поддержка жестов (сжатие, сдвиг и т.д.).

Ajax

Ajax (Asynchronous Javascript And Xml) — технология обращения к серверу без перезагрузки страницы. Обеспечивает асинхронное взаимодействие с сервером, позволяя элементам на странице обновлять данные с сервера, не перезагружая всю страницу. Это новшество стало возможным благодаря введению объекта XMLHttpRequest в начале 2000-х годов и возвестило начало эры "Web 2.0".

Базовая концепция Ajax проста: код JavaScript на стороне браузера программно осуществляет HTTP-запросы к серверу, который возвращает данные, обычно в формате JSON (с которым намного проще работать в JavaScript, чем с XML). Эти данные используются для обеспечения функциональных возможностей в браузере. Хотя в Ajax используется протокол HTTP (точно так же, как и для пересылки веб-страниц без использования Ajax), накладные затраты на передачу и визуализацию страницы снижаются. Это позволяет веб-приложениям выполняться намного быстрее, или по крайней мере выглядеть так с точки зрения пользователя.

Чтобы использовать Ajax, необходим *сервер*. Давайте напишем чрезвычайно простой сервер в Node.js (это тема главы 20), который предоставляет доступ к *конечной*

точке (endpoint) Ajax (особой службе, которая может использоваться в других службах или приложениях). Создайте файл `ajaxServer.js`.

```
const http = require('http');

const server = http.createServer(function(req, res) {
    res.setHeader('Content-Type', 'application/json');
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.end(JSON.stringify({
        platform: process.platform,
        nodeVersion: process.version,
        uptime: Math.round(process.uptime()),
    }));
});

const port = 7070;
server.listen(port, function() {
    console.log('Ajax server started on port ${port}');
});
```

Это код очень простого сервера, который сообщает свою платформу ("linux", "darwin", "win32" и т.д.), версию `Node.js` и продолжительность непрерывной работы сервера.

Благодаря Ajax создается брешь в системе безопасности сайта, называемая *кросс-доменными запросами* (Cross-Origin Resource Sharing — CORS). В этом примере мы добавляем заголовок `Access-Control-Allow-Origin` со значением `*`, который уведомляет клиента (браузер) о том, что не нужно предотвращать вызов из соображений безопасности. На реальном сервере обычно используется тот же самый протокол, домен и порт (который разрешен по умолчанию), либо явно указывается, какой протокол, домен и порт может обращаться к конечной точке. Тем не менее в демонстрационных целях, лучше всего отключить CORS.

Для запуска этого сервера достаточно выполнить команду

```
$ babel-node ajaxServer.js
```

Загрузив страницу `http://localhost:7070` в браузер, вы увидите вывод сервера. Теперь, когда имеется сервер, добавим код Ajax в наш пример HTML-страницы (вы можете использовать ту же страницу, что и ранее в этой главе). Для начала добавим где-нибудь в теле документа элемент, в котором будет отображаться информация.

```
<div class="serverInfo">
    Сервер на платформе <span data-replace="platform">???</span>,
    версия Node <span data-replace="nodeVersion">???</span>. Время
```

```
    его непрерывной работы <span data-replace="uptime">???</span> секунд.
</div>
```

Теперь, когда есть элемент для отображения данных, поступающих с сервера, мы можем использовать объект `XMLHttpRequest`, чтобы выполнить Ajax-запрос. Внизу вашего HTML-файла (прямо перед закрывающим дескриптором `</body>`), добавьте следующий код.

```
<script type="application/javascript;version=1.8">
    function refreshServerInfo() {
    const req = new XMLHttpRequest();
        req.addEventListener('load', function() {
            // TODO: внести эти данные в HTML-код
            console.log(this.responseText);
        });
        req.open('GET', 'http://localhost:7070', true);
        req.send();
    }
    refreshServerInfo();
</script>
```

Этот сценарий осуществляет простой Ajax-запрос. Сначала мы создаем новый объект `XMLHttpRequest`, а затем добавляем обработчик, который перехватывает событие `load` (он будет запущен, если Ajax-запрос успешно завершится). В нем мы пока что только выводим ответ сервера (который находится в свойстве `this.responseText`) на консоль. Затем происходит вызов метода `open`, который фактически устанавливает соединение с сервером. Мы определяем, что это HTTP-запрос `GET`. Это такой же тип запроса, который используется в браузере при посещении вебстраницы (есть и другие типы запросов: `POST`, `DELETE` и др.). Далее мы сообщаем методу URL сервера. И наконец происходит вызов метода `send`, который фактически выполняет запрос. В этом примере мы явно не посылаем данные на сервер, хотя могли бы это сделать.

Запустив этот пример, вы увидите данные, возвращаемые с сервера и отображаемые на консоли. Наш следующий шаг — поместить эти данные в наш HTML-код. Мы структурировали свой HTML так, чтобы проще было найти любой элемент, у которого есть атрибут данных `replace`, и заменить содержимое этого элемента данными из возвращенного объекта. Для этого мы перебираем свойства, которые были возвращены сервером (с использованием метода `Object.keys`), и если есть какие-нибудь элементы с соответствующими атрибутами данных `replace`, мы заменяем их содержимое.

```
req.addEventListener('load', function() {
    // this.responseText - это строка, содержащая JSON; мы используем
    // JSON.parse, чтобы преобразовать ее в объект
    const data = JSON.parse(this.responseText);
```

```
    // В этом примере мы только хотим заменить текст в пределах <div>,
    // имеющего класс "serverInfo"
    const serverInfo = document.querySelector('.serverInfo');

    // Перебор по ключам в объекте, возвращенном с сервера
    // ("platform", "nodeVersion" и "uptime"):
    Object.keys(data).forEach(p => {
        // Найти элементы для замены для этого свойства (если есть)
        const replacements =
            serverInfo.querySelectorAll('[data-replace="${p}"]');
        // заменить все элементы значением, возвращенным с сервера
        for(let r of replacements) {
            r.textContent = data[p];
        }
    });
});
```

Поскольку `refreshServerInfo` — это функция, мы можем вызвать ее в любое время. В частности мы можем обновлять информацию, полученную с сервера периодически (вот почему мы добавили поле `uptime`). Например, если мы хотим обновлять информацию с сервера пять раз в секунду (каждые 200 мс), то можем добавить следующий код.

```
setInterval(refreshServerInfo, 200);
```

Сделав это, мы будем наблюдать в браузере последовательное увеличение времени непрерывной работы сервера!

В этом примере, когда страница загружается впервые, в элементе `<div class=".serverInfo">` находится текстовый заполнитель в виде вопросительных знаков. При медленном соединении с Интернетом пользователь может увидеть эти вопросительные знаки на мгновение, прежде чем они будут заменены информацией, полученной от сервера. Это известная проблема *появления нестилизованного содержимого* (Flash Of Unstyled Content — FOUC). Одно из ее решений подразумевает получение от сервера начальной страницы с правильными значениями. Другое решение — полностью скрыть элемент, пока его содержимое не будет изменено. Это может вызывать раздражение у пользователей, но зачастую это куда лучше, чем созерцание бессмысленных вопросительных знаков.

В этом разделе были рассмотрены только основные концепции, используемые при создании Ajax-запросов. Более подробная информация по этой теме приведена в статье "*Using XMLHttpRequest*" библиотеки MDN.

Заключение

Как вы уже наверное заметили, в этой главе, при создании веб-приложения, кроме самого языка JavaScript, мы использовали несколько сложных технологий. Здесь мы затронули их только поверхностно, и если вы — веб-разработчик, я рекомендую книгу Сэмми Пьюривала (Semmy Purewal) *Основы разработки веб-приложений (Learning Web App Development)*, а если вы хотите узнать больше о CSS, то любую из книг Эрика А. Мейера (Eric A. Meyer).

Библиотека jQuery

jQuery — это популярная библиотека для манипулирования элементами DOM и выполнения Ajax-запросов. Библиотека jQuery не может сделать ничего, что вы не смогли бы сделать с API DOM (в конце концов, jQuery сама основана на API DOM), но она предоставляет три основных преимущества.

- jQuery избавляет от необходимости заботься об индивидуальных особенностях различных браузеров, реализующих API DOM (особенно устаревших).
- jQuery предоставляет упрощенный API Ajax (что очень кстати, поскольку на нынешних веб-сайтах использовать Ajax сложно).
- jQuery предоставляет множество мощных и компактных расширений для встроенного API DOM.

Сегодня наблюдается рост сообщества веб-разработчиков, полагающих, что в jQuery больше нет необходимости, поскольку API DOM и современные браузеры достигли совершенства. Это сообщество рекламирует эффективность и чистоту "традиционного JavaScript". Это правда, что первый пункт (особенности браузера) со временем становится менее актуальным, но полностью он не снимается. Я полагаю, что библиотека jQuery до сих пор остается актуальной и предоставляет много средств, повторная реализация которых с помощью API DOM отняла бы чрезвычайно много времени. Решите вы использовать jQuery или нет, ее высокая популярность требует от квалифицированного веб-разработчика знания хотя бы ее основ.

Всемогущий доллар (знак)

jQuery была одной из первых библиотек, в которых использовалось включение в JavaScript знака доллара как идентификатора. Возможно, первоначально это решение принималось из-за оригинальности разработчиков, но сейчас, благодаря вездесущности jQuery, оно оказалось поистине пророческим. Включив jQuery в свой проект, вы можете использовать либо переменную `jQuery`, либо намного более краткий псевдоним `$`.[1] Здесь мы будем использовать псевдоним `$`.

[1] Можно запретить jQuery использовать псевдоним `$`, если это вступает в противоречие с другой библиотекой (см. *jQuery.noConflict*).

Подключение jQuery

Самый простой способ подключения библиотеки jQuery — это использовать сеть CDN.

```
<script src="//code.jquery.com/jquery-2.1.4.min.js"></script>
```

В jQuery 2.x уже не поддерживаются устаревшие браузеры Internet Explorer 6, 7 и 8. Если необходима поддержка для этих браузеров, используйте jQuery 1.x. Библиотека jQuery 2.x значительно меньше и проще, поскольку не должна поддерживать эти устаревшие браузеры.

Ожидание загрузки и построения дерева DOM

Способ, которым браузер читает, интерпретирует и визуализирует HTML-файл, довольно сложен, и многие веб-разработчики по неосторожности сталкиваются с неприятностями при попытке программного доступа к элементам DOM прежде, чем у браузера будет шанс их загрузить.

jQuery позволяет вам поместить свой код в функцию обратного вызова, которая будет вызвана, как только браузер полностью загрузит страницу и построит дерево DOM.

```
$(document).ready(function() {
    // расположенный здесь код запускается после загрузки
    // всего HTML и построения дерева DOM
});
```

Эту методику можно вполне безопасно использовать многократно, что позволяет помещать код jQuery в различные места и все еще иметь безопасное ожидание построения дерева DOM. Есть также сокращенная версия, которая эквивалентна предыдущей.

```
$(function() {
    // расположенный здесь код запускается после загрузки
    // всего HTML-документа и построения дерева DOM
});
```

Помещение всего кода в такой блок является общепринятой практикой при использовании библиотеки jQuery.

Элементы DOM в оболочке jQuery

Основная методика манипулирования DOM с использованием jQuery — это *элементы DOM в оболочке jQuery* (jQuery-wrapped DOM elements). Любая манипуляция DOM, осуществляемая с использованием jQuery, начинается с создания объекта jQuery, являющегося "оболочкой" для набора элементов DOM (имейте в виду, что набор может быть пустым или содержать только один элемент).

Функция jQuery (`$` или `jQuery`) создает набор элементов DOM в оболочке jQuery (который начиная с этого момента мы будем называть "объект jQuery" (jQuery object); просто помните, что объект jQuery содержит набор элементов DOM!). Функцию jQuery главным образом вызывают одним из двух способов: с селектором CSS или с HTML-кодом.

Вызов jQuery с селектором CSS возвращает объект jQuery, соответствующий этому селектору (подобно возвращению из `document.querySelectorAll`). Например, чтобы получить объект jQuery, соответствующий всем дескрипторам `<p>`, просто сделайте так.

```
const $paras = $('p');
$paras.length;       // количество соответствующих дескрипторов абзаца
typeof $paras;             // "object"
$paras instanceof $;       // true
$paras instanceof jQuery; // true
```

Вызов jQuery с HTML-кодом, напротив, создает новые элементы DOM на основании HTML, который вы предоставляете (подобно тому, как это происходит при установке значения свойства `innerHTML` элемента).

```
const $newPara = $('<p>Только что созданный абзац...</p>');
```

Обратите внимание, что имя переменной, которой мы присваиваем объект jQuery, в обоих примерах начинается со знака доллара. Необходимости в этом нет, но следовать этому соглашению имеет смысл. Так можно быстро распознавать переменные, которые являются объектами jQuery.

Манипулирование элементами

Теперь, когда мы имеем кой-какую информацию об объектах jQuery, что мы можем сделать с ними? jQuery существенно упрощает добавление и удаление содержимого. Наилучший способ рассмотрения этих примеров — загрузить HTML-пример в браузер и запустить эти примеры на консоли. Будьте готовы перезагрузить файл, поскольку мы будем произвольно удалять, добавлять и изменять его содержимое.

jQuery предоставляет методы `text` и `html`, которые грубо говоря являются эквивалентами присвоения свойствам `textContent` и `innerHTML` элемента DOM. Например, чтобы заменить каждый абзац в документе одним и тем же текстом, используем

```
$('p').text('ВСЕ АБЗАЦЫ ЗАМЕНЕНЫ');
```

Аналогично мы можем применить метод `html`, чтобы использовать HTML-код.

```
$('p').html('<i>ВСЕ</i> АБЗАЦЫ ЗАМЕНЕНЫ');
```

Из этого следует важный момент: jQuery существенно упрощает работу со многими элементами сразу. При использовании API DOM метод `document.`

querySelectorAll() возвратит несколько элементов, но их перебор и выполнение любых необходимых операций — это уже забота программиста. Библиотека jQuery сама осуществляет перебор и стандартно подразумевает, что вы хотите выполнить действия с каждым элементом в объекте jQuery. Что если вы хотите изменить только третий абзац? jQuery предоставляет метод eq, который возвращает новый объект jQuery, содержащий одиночный элемент.

```
$('p')        // соответствует всем абзацам
    .eq(2)    // третий абзац (отсчитывается от нуля)
    .html('<i>ТРЕТИЙ</i> АБЗАЦ ЗАМЕНЕН');
```

Чтобы удалить элементы, достаточно вызвать метод remove объекта jQuery. Следующее удалит все абзацы.

```
$('p').remove();
```

Это демонстрирует еще одну важную парадигму в разработке jQuery: *сцепление* (chaining). Все методы jQuery возвращают объект jQuery, что позволяет сцеплять их вызовы, как мы сделали только что. Сцепление обеспечивает очень мощный и компактный синтаксис для манипулирования несколькими элементами.

Библиотека jQuery предоставляет много методов для добавления нового содержимого. Один из этих методов, append, просто добавляет предоставленное содержимое в каждый элемент в объекте jQuery. Например, можно очень легко добавить сноску к каждому абзацу.

```
$('p')
    .append('<sup>*</sup>');
```

Метод append добавляет дочерний элемент к соответствующим элементам; мы также можем вставить элементы одного уровня, используя методы before или after. Например, добавим горизонтальные линии (элементы <hr>) перед каждым абзацем и после него.

```
$('p')
    .after('<hr>')
    .before('<hr>');
```

Эти методы вставки также имеют соответствующие дубликаты appendTo, insertBefore и insertAfter, меняющие порядок вставки на обратный, что может быть полезно в определенных ситуациях, например таких.

```
$('<sup>*</sup>').appendTo('p'); // эквивалент $('p').append('<sup>*</sup>')
$('<hr>').insertBefore('p');     // эквивалент $('p').before('<hr>')
$('<hr>').insertAfter('p');      // эквивалент $('p').after('<hr>');
```

Библиотека jQuery также весьма упрощает изменение стиля элемента. Вы можете добавлять класс, используя метод addClass, удалять класс, используя removeClass, и *переключать* классы, используя toggleClass (что добавит класс, если у элемента

его нет, и удалит класс, если он уже есть). Вы можете также манипулировать стилем непосредственно, используя метод `css`. Мы также познакомимся с селекторами `:even` и `:odd`, которые позволяют выбрать любой элемент. Например, если бы нам нужно было выделить каждый нечетный абзац красным цветом, то могли бы сделать следующее.

```
$('p:odd').css('color', 'red');
```

Сцепление в jQuery не является чем-то обязательным для выбора подмножества соответствующих элементов. Мы уже видели метод `eq`, который позволяет свести объект jQuery к одному элементу; для модификации выбранных элементов мы можем также использовать методы `filter`, `not` и `find`. Метод `filter` сводит набор к элементам, которые соответствуют определенному селектору. Например, мы можем использовать `filter` в цепочке, чтобы выделить каждый нечетный абзац красным цветом *после того, как* он будет изменен.

```
$('p')
    .after('<hr>')
    .append('<sup>*</sup>')
    .filter(':odd')
    .css('color', 'red');
```

Метод `not` — это, по существу, метод, обратный методу `filter`. Например, можно добавить `<hr>` после каждого абзаца, а затем сдвинуть все абзацы, которым не назначен класс `highlight`.

```
$('p')
    .after('<hr>')
    .not('.highlight')
    .css('margin-left', '20px');
```

И наконец метод `find` возвращает набор *дочерних* элементов, которым соответствует данный запрос (в противоположность методу `filter`, который фильтрует существующий набор). Например, мы можем добавить `<hr>` перед каждым абзацем, а затем увеличить размер шрифта элементов, которым назначен класс `code`, который в нашем примере является потомком абзацев.

```
$('p')
    .before('<hr>')
    .find('.code')
    .css('font-size', '30px');
```

Извлечение объектов jQuery из оболочки

"Распаковать" объект jQuery (получить доступ к внутренним элементам DOM) можно, используя метод `get`. Давайте получим элемент DOM для второго абзаца.

```
const para2 = $('p').get(1); // второй <p> (начиная с нуля)
```

Получим массив, содержащий все элементы DOM абзацев.

```
const paras = $('p').get();  // массив всех элементов <p>
```

Ajax

jQuery предоставляет удобные методы, упрощающие Ajax-запросы. В jQuery реализован метод `ajax`, обеспечивающий полный контроль над Ajax-запросами. Он предоставляет также методы `get` и `post`, которые выполняют Ajax-запросы самых распространенных типов. Хотя эти методы поддерживают обратные вызовы, они также возвращают обязательства, которые и являются рекомендуемым средством обработки ответа сервера. Например, мы можем использовать метод `get`, чтобы переписать наш пример `refreshServerInfo` так.

```
function refreshServerInfo() {
    const $serverInfo = $('.serverInfo');
    $.get('http://localhost:7070').then(
        // успешное возвращение
        function(data) {
            Object.keys(data).forEach(p => {
                $('[data-replace="${p}"]').text(data[p]);
            });
        },
        function(jqXHR, textStatus, err) {
            console.error(err);
            $serverInfo.addClass('error')
                .html('Ошибка при подключении к серверу.');
        }
    );
}
```

Как можно заметить, использование jQuery значительно упростило наш код Ajax.

Заключение

Будущее jQuery неясно. Не приведут ли усовершенствования API JavaScript и браузера к устареванию jQuery? Не победят ли борцы за "чистоту JavaScript"? Только время покажет. Я чувствую, что библиотека jQuery будет полезной и в обозримом будущем. Конечно, использование jQuery остается весьма популярным, и любой стремящийся к успеху разработчик должен знать по крайней мере ее основы.

Если вы хотите узнать больше о jQuery, я рекомендую книгу Адама Фримена *jQuery 2.0 для профессионалов* (пер. с англ, ИД "Вильямс", 2016, ISBN 978-5-8459-1919-9). Сетевая документация по jQuery также очень хороша.

ГЛАВА 20

Платформа Node

Вплоть до 2009 года JavaScript был почти исключительно языком сценариев для браузеров.[1] В 2009 году разработчик компании Joyent по имени Райан Дал (Ryan Dahl), расстроенный из-за состояния серверных опций, создал Node. Платформа Node была принята молниеносно и стала популярной даже на достаточно консервативном корпоративном рынке.

Тем, кому JavaScript понравился как язык, платформа Node позволила использовать его для задач, традиционно связанных с другими языками. Для веб-разработчиков привлекательность оказалась куда сильнее, чем просто выбор языка. Возможность писать серверный код на JavaScript означает *единообразную среду* программирования. Больше не нужно в уме переключать контексты выполнения программ, вам не нужны специалисты по другим серверным технологиям и (что, возможно, важнее всего) один и тот же код можно запускать как на сервере, так и на клиенте.

Хотя платформа Node была предназначена для того, чтобы сделать возможной разработку веб-приложений, ее перенос на сервер неожиданно обеспечил другое нетрадиционное использование, такое как разработка приложений для рабочего стола и системных сценариев. В некотором смысле платформа Node позволила JavaScript повзрослеть и укрепиться.

Основные принципы Node

Написание приложений для Node ничем не отличается от написания любых других приложений на JavaScript. Я не хочу сказать, что вы можете просто взять любую JavaScript-программу для браузера и запустить ее в среде Node, поскольку в коде JavaScript для браузера используется API, специфичное для браузера. В частности, в Node нет никакого DOM (зачем он нужен, ведь никакого HTML-документа нет и в помине!). Аналогично в Node есть свой интерфейс API, который специфичен для Node и не поддерживается в браузере. Некоторые вещи, такие как прямые

[1] Попытки создания серверного JavaScript предпринимались и до Node; в частности, Netscape Enterprise Server поддерживал серверный JavaScript уже в 1995 году. Однако серверный JavaScript не получил распространения до 2009 года, когда появилась Node.

вызовы функций операционной и файловой системы, недоступны в браузере из соображений безопасности (можете себе представить, что сделали бы хакеры, если бы они смогли получить доступ к вашим файлам прямо из браузера?). Другие возможности, такие как создание веб-сервера, для браузера просто бесполезны.

Важно понять, что составляет основу *JavaScript* и что является *частью API*. Программист, который всегда писал код для браузера, мог бы вполне резонно полагать, что объекты `window` и `document` — это просто часть JavaScript. Однако это части *API*, предоставляемых средой браузера (как было описано в главе 18). В этой главе мы будем рассматривать API, предоставляемые платформой Node.

Если это еще не сделано, установите среду Node и npm (см. главу 2).

Модули

Модули — это механизм для упаковки кода и применения в нем пространств имен. *Пространства имен* (namespacing) — это средство для предотвращения *конфликтов имен*. Например, если Аманда и Тайлер написали два варианта функции `calculate`, а вы просто берете и копируете их код в свою программу, то вторая функция заменит первую. Пространства имен позволяют тем или иным образом обращаться к функции `calculate` "от Аманды" и к функции `calculate` "от Тайлера". Давайте рассмотрим, как модули Node решают эту проблему. Создайте файл `amanda.js`.

```
function calculate(a, x, n) {
    if(x === 1) return a*n;
    return a*(1 - Math.pow(x, n))/(1 - x);
}

module.exports = calculate;
```

И создайте файл `tyler.js`.

```
function calculate(r) {
    return 4/3*Math.PI*Math.pow(r, 3);
}

module.exports = calculate;
```

Вполне понятно, что Аманда и Тайлер не проявили фантазии при выборе имен своих функций, но для примера мы позволим этому непотребству продолжиться. Важная строка в обоих этих файлах — `module.exports = calculate;`. Конструкция `module` — это специальный объект, который введен в Node ради реализации модулей. Все то, что будет присвоено его свойству `exports`, будет *экспортироваться* из этого модуля. Теперь, написав несколько модулей, давайте посмотрим, как их можно

использовать в совершенно другой программе. Давайте создадим файл `app.js`, в котором *импортируем* эти модули.

```
const amanda_calculate = require('./amanda.js');
const tyler_calculate = require('./tyler.js');
console.log(amanda_calculate(1, 2, 5)); // выводит 31
console.log(tyler_calculate(2));        // выводит 33.510321638291124
```

Обратите внимание, что выбранные нами имена (`amanda_calculate` и `tyler_calculate`) совершенно произвольны; это только переменные. Получаемые значения являются результатом вызова функции Node `require`.

Математически подготовленный читатель уже, вероятно, узнал эти два вычисления: Аманда предоставляет сумму геометрической прогрессии $a + ax + ax^2 + \ldots + ax^{n-1}$, а Тайлер вычисляет объем сферы радиусом r. Теперь, когда известно, что это такое, мы можем упрекнуть Аманду и Тайлера за плохой выбор имен и выбрать более подходящие имена в `app.js`.

```
const geometricSum = require('./amanda.js');
const sphereVolume = require('./tyler.js');

console.log(geometricSum(1, 2, 5)); // выводит 31
console.log(sphereVolume(2));       // выводит 33.510321638291124
```

Модули могут экспортировать значение любого типа (даже базового, хотя на это есть немного причин). Обычно ваш модуль будет содержать не одну функцию, а несколько. В этом случае вы должны экспортировать объект со свойствами функции. Предположим, что Аманда — математик, которая снабжает нас многими другими полезными алгебраическими функциями, а не только функцией расчета суммы геометрической прогрессии.

```
module.exports = {
    geometricSum(a, x, n) {
        if(x === 1) return a*n;
        return a*(1 - Math.pow(x, n))/(1 - x);
    },
    arithmeticSum(n) {
        return (n + 1)*n/2;
    },
    quadraticFormula(a, b, c) {
        const D = Math.sqrt(b*b - 4*a*c);
        return [(-b + D)/(2*a), (-b - D)/(2*a)];
    },
};
```

Это приводит к более традиционному подходу к пространствам имен — мы присваиваем имя тому, что возвращается из модуля, но возвращаемое значение (объект) содержит собственные имена.

```
const amanda = require('./amanda.js');
console.log(amanda.geometricSum(1, 2, 5));        // выводит 31
console.log(amanda.quadraticFormula(1, 2, -15)); // выводит [ 3, -5 ]
```

Здесь нет никакого чуда: модуль просто экспортирует обычный объект с функциональными свойствами (не позволяйте сокращенному синтаксису ES6 себя запутать; это обычные функции, являющиеся свойствами объекта). Данная парадигма настолько распространена, что для нее был предусмотрен сокращенный синтаксис, использующий специальную переменную `exports`. Мы можем переписать экспортируемый код Аманды более компактно.

```
exports.geometricSum = function(a, x, n) {
    if(x === 1) return a*n;
    return a*(1 - Math.pow(x, n))/(1 - x);
};

exports.arithmeticSum = function(n) {
    return (n + 1)*n/2;
};

exports.quadraticFormula = function(a, b, c) {
    const D = Math.sqrt(b*b - 4*a*c);
    return [(-b + D)/(2*a), (-b - D)/(2*a)];
};
```

Сокращение "`exports`" работает только при экспорте объектов; если вы хотите экспортировать функцию или некое другое значение, используйте `module.exports`. Кроме того, вы не можете их смешивать: используйте либо одно, либо другое.

Базовые, файловые и npm-модули

Модули относятся к трем категориям: *базовые* (core module), *файловые* (file module) и *npm-модули* (npm module). Имена базовых модулей зарезервированы; эти модули, например, `fs` и `os`, предоставляет сама среда Node (мы их обсудим далее в этой главе). С файловыми модулями мы уже встречались, когда создавали файл с экспортируемой функцией, в котором присваивалось нечто свойству `module.exports`, а затем использовали этот файл в других программах. Модули npm — это обычные файловые модули, которые находятся в специальной папке, называемой `node_modules`. Когда вы используете функцию `require`, Node определяет тип модуля (их описание приведены в табл. 20.1) из передаваемой строки.

Таблица 20.1. Типы модулей

Тип	Строка, передаваемая `require`	Примеры

Базовый	Не начинается с /, ./ или ../	`require('fs')`
		`require('os')`
		`require('http')`
		`require('child_process')`

Файловый	Начинается с /, ./ или ../	`require('./debug.js')`
		`require('/full/path/to/module.js')`
		`require('../a.js')`
		`require('../../a.js')`

npm	Не базовый модуль и не начинается с /, ./ или ../	`require('debug')`
		`require('express')`
		`require('chalk')`
		`require('koa')`
		`require('q')`

Некоторые базовые модули, такие как `process` и `buffer`, являются глобальными. Они доступны всегда и не требуют явного оператора `require`. Базовые модули приведены в табл. 20.2.

Таблица 20.2. Базовые модули

Модуль	Глобальный	Описание
`assert`	Нет	Используется в проверочных целях
`buffer`	Да	Используется для операций ввода-вывода (I/O) (прежде всего, в файл и сеть)
`child_process`	Нет	Функции для запуска внешних программ (Node и др.)
`cluster`	Нет	Позволяет использовать несколько процессов для повышения производительности
`crypto`	Нет	Встроенные криптографические библиотеки
`dns`	Нет	Функции системы доменных имен (DNS) для преобразования сетевых имен
`domain`	Нет	Позволяет группировать ввод-вывод и другие асинхронные операции для изоляции ошибок
`events`	Нет	Утилиты для поддержки асинхронных событий
`fs`	Нет	Операции файловой системы
`http`	Нет	Сервер HTTP и связанные с ним утилиты
`https`	Нет	Сервер HTTPS и связанные с ним утилиты
`net`	Нет	Асинхронное сетевое API на базе сокетов
`os`	Нет	Утилиты операционной системы
`path`	Нет	Утилиты имен и путей файловой системы
`punycode`	Нет	Кодировка символов Unicode с помощью ограниченного подмножества символов ASCII
`querystring`	Нет	Утилиты для анализа и создания строк запросов URL

Окончание табл. 20.2

Модуль	Глобальный	Описание
`readline`	Нет	Интерактивные утилиты ввода-вывода; в первую очередь, для программ командной строки
`smalloc`	Нет	Обеспечивает явное распределение памяти для буферов
`stream`	Да	Передача потоковых данных
`string_decoder`	Нет	Преобразование буфера в строки
`tls`	Нет	Утилиты TLS (Transport Layer Security — безопасный транспортный уровень)
`tty`	Нет	Низкоуровневые функции TTY(TeleTYpewriter)
`dgram`	Нет	Утилиты UDP (User Datagram Protocol — протокол пользовательских дейтаграмм) для работы в сети
`url`	Да	Утилиты анализа URL
`util`	Нет	Внутренние утилиты Node
`vm`	Нет	Виртуальная машина (JavaScript): обеспечивает функции метапрограммирования и создания контекста
`zlib`	Нет	Утилита сжатия

Рассмотрение всех этих модулей выходит за рамки данной книги. Мы обсудим лишь самые важные из них, но это даст вам отправную точку для получения дополнительной информации. Подробная документация для этих модулей доступна в *документации API Node* (`https://nodejs.org/api/`).

И наконец, есть *npm-модули* — файловые модули со специфическим соглашением об именовании. Если вам необходим некоторый модуль `x` (где `x` — не базовый модуль), то Node будет искать в текущем каталоге подкаталог `node_modules`. Если он его найдет, то будет искать модуль `x` в этом каталоге. Если он его не найдет, то перейдет к родительскому каталогу, и снова начнет искать каталог `node_modules` и продолжит поиск в нем. Процесс будет повторяться, пока не будет найден модуль или достигнут корневой каталог. Например, если ваш проект находится в каталоге `/home/jdoe/test_project` и в своем файле приложения вы вызываете функцию `require('x')`, Node будет искать модуль *x* в перечисленных ниже каталогах в таком порядке.

- `/home/jdoe/test_project/node_modules/x`
- `/home/jdoe/node_modules/x`
- `/home/node_modules/x`
- `/node_modules/x`

Для большинства проектов создается один каталог `node_modules` в корневом каталоге приложений. Кроме того, вы не должны ничего добавлять или удалять из того каталога вручную; позвольте утилите `npm` сделать все самостоятельно. Однако весьма

полезно знать, как Node ищет импортируемые модули, особенно когда наступает время поиска проблемы в модулях стороннего производителя.

Не помещайте те модули, которые вы пишете сами, в каталог `node_modules`. Работать это будет, но особенность каталога `node_modules` в том, что он может быть удален утилитой `npm` в любой момент и воссоздан из зависимостей, перечисленных в файле `package.json` (см. главу 2).

Вы можете, конечно, опубликовать собственный модуль с помощью утилиты `npm` и управлять им, используя `npm`, но тогда вы должны избегать внесения в него изменений непосредственно в каталоге `node_modules`!

Изменение параметров модулей с помощью модулей-функций

Обычно модули экспортируют объекты, а иногда и одну функцию. Однако есть и другой популярный сценарий их использования — модуль, который экспортирует функцию, предназначенную для немедленного вызова. Речь идет о том, что возвращаемое модулем значение само по себе является функцией, которая сразу же вызывается. Другими словами, вы не используете далее в своей программе то, что возвращает модуль, а вызываете эту функцию и используете то, что она возвращает. Этот сценарий используется, когда нужно изменить параметры модуля либо получить информацию об окружающем контексте. Давайте рассмотрим реальный пакет npm `debug`. При импортировании функции `debug` ей передается текстовая строка, которая будет использоваться как префикс при выводе отладочных сообщений. Благодаря этому можно будет различать сообщения, выведенные из различных частей программы.

```
const debug = require('debug')('main'); // обратите внимание, что мы
                                        // сразу вызываем функцию, которую
                                        // возвращает модуль

debug("начало");                        // выводит
                                        // "main начало +0ms", если
                                        // отладка будет разрешена
```

Чтобы разрешить отладку с использованием библиотеки `debug`, установите переменную окружения `DEBUG`. Для нашего примера мы установили бы `DEBUG=main`. Вы можете также установить `DEBUG=*`, что разрешает вывод всех отладочных сообщений.

Из этого примера ясно, что модуль `debug` возвращает функцию (поскольку мы непосредственно вызываем это как функцию)... и эта функция сама возвращает функцию, которая "помнит" строку, переданную первой функции. По сути мы

"интегрировали" значение в этот модуль. Давайте посмотрим, как мы могли бы реализовать собственный модуль debug.

```
let lastMessage;

module.exports = function(prefix) {
    return function(message) {
        const now = Date.now();
        const sinceLastMessage = now - (lastMessage || now);
        console.log('${prefix} ${message} +${sinceLastMessage}ms');
        lastMessage = now;
    }
}
```

Этот модуль экспортирует функцию, которая написана так, чтобы значение переменной prefix можно было использовать в модуле. Обратите внимание, что у нас есть и другое значение, lastMessage, которое является временной меткой последнего сообщения, которое было выведено; мы используем его, чтобы вычислить время между сообщениями.

Этот фрагмент кода демонстрирует важный момент: что произойдет, если вы импортируете модуль многократно? Рассмотрим, например, что происходит при импорте самодельного модуля debug дважды.

```
const debug1 = require('./debug')('Первый');
const debug2 = require('./debug')('Второй');

debug1('запущен первый отладчик!')
debug2('запущен второй отладчик!')

setTimeout(function() {
    debug1('прошло немного времени...');
    debug2('что случилось?');
}, 200);
```

Если вы ожидаете увидеть нечто такое:

```
Первый запущен первый отладчик! +0ms
Второй запущен второй отладчик! +0ms
Первый прошло немного времени... +200ms
Второй что случилось? +200ms
```

то я вас разочарую. На самом деле вы увидите это (плюс или минус несколько миллисекунд).

```
Первый запущен первый отладчик! +0ms
Второй запущен второй отладчик! +0ms
Первый прошло немного времени... +200ms
Второй что случилось? +0ms
```

Оказывается, Node импортирует каждый конкретный модуль только один раз при запуске приложения Node. Таким образом, даже при том, что мы импортируем свой модуль `debug` дважды, Node "помнит", что мы уже это импортировали раньше, и использует тот же экземпляр. Таким образом, даже при том, что `debug1` и `debug2` — это отдельные функции, в их обеих используется ссылка на одну и ту же переменную `lastMessage`.

Такое поведение программы вполне безопасно и желательно. По соображениям производительности, экономии памяти и удобства сопровождения, модули должны загружаться только один раз!

Сценарий импортирования, который мы использовали при создании нашего самодельного модуля `debug`, подобен тому, что используется и для его npm-тезки. Но если нам действительно нужно получить несколько журналов отладки с независимым хронометражем, то переменную `lastMessage`, хранящую временную метку, нужно переместить в тело функции, которую возвращает модуль. Тогда каждый раз при создании регистратора ей будет присваиваться новое, независимое значение.

Доступ к файловой системе

Во многих книгах по программированию *доступ к файловой системе* рассматривается с самого начала, поскольку это критически важная часть "обычного" программирования. Бедный JavaScript: вплоть до появления Node он не был членом клуба файловой системы.

В примерах из этой главы подразумевается, что корневой каталог вашего проекта `/home/<jdoe>/fs`. Это типичный путь к каталогу в системах Unix, только нужно вместо *<jdoe>* подставить имя вашей учетной записи. Те же принципы применимы и к системе Windows (в которой корневой каталог вашего проекта мог бы располагаться в папке `C:\Users\<John Doe>\Documents\fs`).

Чтобы создать файл, используйте метод `fs.writeFile`. Создайте в корневом каталоге своего проекта файл `write.js`.

```
const fs = require('fs');

fs.writeFile('hello.txt', 'Привет из Node!', function(err) {
    if(err) return console.log('Ошибка при записи в файл.');
});
```

Этот код создаст файл `hello.txt` в том каталоге, в котором вы находились при запуске приложения `write.js`. Здесь подразумевается, что вы имеете права доступа по записи к этому каталогу и что в каталоге нет заранее созданного файла только

для чтения hello.txt. Каждый раз при запуске приложения Node, оно будет использовать *текущий рабочий каталог*, который может отличаться от того, где расположен сам файл, как показано ниже.

```
$ cd /home/jdoe/fs
$ node write.js     # текущий рабочий каталог /home/jdoe/fs
                    # создается /home/jdoe/fs/hello.txt

$ cd ..             # теперь текущий рабочий каталог /home/jdoe
$ node fs/write.js # создается /home/jdoe/hello.txt
```

В Node существует специальная переменная, __dirname, которая всегда содержит путь к каталогу, в котором располагается файл исходного кода. Например, мы можем изменить свой пример так.

```
const fs = require('fs');

fs.writeFile(__dirname + '/hello.txt',
        'Привет из Node!', function(err) {
    if(err) return console.error('Ошибка при записи в файл.');
});
```

Теперь приложение write.js всегда будет создавать файл hello.txt в каталоге /home/<jdoe>/fs (где находится write.js). Использование конкатенации строк для объединения переменной __dirname и нашего имени файла — не очень хорошая идея. Это может вызвать проблемы на компьютере под управлением Windows, поскольку разделитель имен каталогов там другой. Поэтому в Node в модуле path предусмотрены независимые от платформы утилиты для работы с именами файлов и путями. Таким образом, мы можем переписать этот модуль так, чтобы он без проблем работал на любой платформе.

```
const fs = require('fs');
const path = require('path');

fs.writeFile(path.join(__dirname, 'hello.txt'),
        'Привет из Node!', function(err) {
    if(err) return console.error('Ошибка при записи в файл.');
});
```

Метод path.join объединяет элементы пути, используя тот разделитель каталогов, который принят в текущей операционной системе, что является хорошей практикой.

Что если мы хотим теперь прочитать содержимое этого файла? Для этого мы используем метод fs.readFile. Создайте файл read.js.

```
const fs = require('fs');
const path = require('path');
```

```
fs.readFile(path.join(__dirname, 'hello.txt'), function(err, data) {
    if(err) return console.error('Ошибка при чтении файла.');
    console.log('Содержимое файла:');
    console.log(data);
});
```

Если вы запустите этот пример, то можете быть неприятно удивлены результатом.

```
Содержимое файла:
< Buffer d0 9f d1 80 d0 b8 d0 b2 d0 b5 d1 82 20 d0 b8 d0 b7 20 4e 6f 64 65 21>
```

Если преобразовать эти шестнадцатеричные коды в их эквивалент ASCII/Unicode, то вы обнаружите, что получится текстовая строка `Привет из Node!`, но наша программа в ее текущем состоянии не очень дружественна. Если при вызове метода `fs.readFile` вы не укажете, какую именно *кодировку* нужно использовать, то он возвратит *буфер*, содержащий "сырые" двоичные данные. Хотя мы не определяли кодировку символов в `write.js` явно, стандартной кодировкой для символьных строк в JavaScript является UTF-8 (кодировка Unicode). Мы можем изменить файл `read.js`, определив UTF-8, и получить ожидаемый результат.

```
const fs = require('fs');
const path = require('path');

fs.readFile(path.join(__dirname, 'hello.txt'),
        { encoding: 'utf8' }, function(err, data) {
    if(err) return console.error('Ошибка при чтении файла.');
    console.log('Содержимое файла:');
    console.log(data);
});
```

У всех функций модуля `fs` есть синхронные эквиваленты (их имена завершаются суффиксом "`Sync`"). В `write.js` мы можем использовать синхронный эквивалент.

```
fs.writeFileSync(path.join(__dirname, 'hello.txt'), 'Привет из Node!');
```

И в `read.js` мы можем использовать синхронный эквивалент.

```
const data = fs.readFileSync(path.join(__dirname, 'hello.txt'),
    { encoding: 'utf8' });
```

Поскольку в синхронных версиях функций обработка ошибок выполняется с использованием исключений, чтобы сделать наши примеры надежнее, заключим их в блоки `try/catch`, например так.

```
try {
    fs.writeFileSync(path.join(__dirname, 'hello.txt'), 'Привет из Node!');
} catch(err) {
    console.error('Ошибка при записи файла.');
}
```

Синхронные функции файловой системы заманчиво удобны. Но если вы пишете веб-сервер или сетевое приложение, помните, что скорость его работы будет максимальной при асинхронном выполнении. В этих случаях всегда следует использовать асинхронные версии функций ввода-вывода. Если вы пишете утилиту командной строки, использование синхронных версий функций обычно не представляет особой проблемы.

Используя метод `fs.readdir`, вы можете вывести список файлов в каталоге. Создайте файл `ls.js`.

```
const fs = require('fs');

fs.readdir(__dirname, function(err, files) {
    if(err) return console.error('Невозможно прочитать содержимое каталога');
    console.log('Содержимое каталога ${__dirname}:');
    console.log(files.map(f => '\t' + f).join('\n'));
});
```

В модуле `fs` содержится довольно много функций файловой системы; вы можете удалять файлы (`fs.unlink`), перемещать или переименовывать их (`fs.rename`), получать информацию о файлах и каталогах (`fs.stat`) и многое другое. Более подробная информация по этой теме приведена в *документации по Node API* (`https://nodejs.org/api/fs.html`).

Переменная `process`

Каждая выполняющаяся программа Node имеет доступ к переменной `process`, которая позволяет получать информацию о выполнении текущего процесса и управлять им. Например, если ваше приложение встречается с ошибкой, настолько серьезной, что продолжение выполнения становится нецелесообразным или бессмысленным (*неустранимая ошибка* (fatal error)), вы можете немедленно остановить ее выполнение, вызвав метод `process.exit`. Можно также передать числовой *код завершения* (exit code), который используется сценариями для определения, успешно ли завершилась программа. Традиционно код завершения 0 означает "отсутствие ошибки", а отличный от нуля код означает ту или иную ошибку. Рассмотрим сценарий, который обрабатывает файлы `.txt` в подкаталоге `data`: если никаких файлов для обработки нет и делать нечего, то программа завершает работу немедленно, но это не ошибка. С другой стороны, если подкаталог `data` не будет существовать, то эта проблема будет считаться более серьезной, и программа должна вернуть код ошибки. Вот как могла бы выглядеть эта программа.

```
const fs = require('fs');

fs.readdir('data', function(err, files) {
    if(err) {
```

```
        console.error("Ошибка: не могу прочитать каталог data.");
        process.exit(1);
    }
    const txtFiles = files.filter(f => /\.txt$/i.test(f));
    if(txtFiles.length === 0) {
        console.log("Файлы .txt не найдены.");
        process.exit(0);
    }
    // обработка файлов .txt...
});
```

Объект `process` также позволяет обращаться к массиву, содержащему *аргументы командной строки*, переданные программе. Запуская приложение Node, вы можете передать ему необязательные аргументы командной строки. Например, мы могли написать программу, которой передаются в виде аргументов командной строки несколько имен файлов, а она выводит количество строк текста в каждом файле. Мы могли бы вызвать программу так.

```
$ node linecount.js file1.txt file2.txt file3.txt
```

Аргументы командной строки содержатся в массиве `process.argv`.[2] Прежде чем подсчитывать строки в наших файлах, давайте выведем значение свойства `process.argv`, чтобы понять, что нам передается.

```
console.log(process.argv);
```

Наряду с `file1.txt`, `file2.txt` и `file3.txt` вы увидите несколько дополнительных элементов в начале массива.

```
[ 'node',
  '/home/jdoe/linecount.js',
  'file1.txt',
  'file2.txt',
  'file3.txt' ]
```

Первый элемент — это *интерпретатор*, или программа, которая интерпретирует файл исходного кода (в данном случае — `node`). Второй элемент — это полный путь к выполняемому сценарию, а остальная часть элементов является всеми переданными программе аргументами. Поскольку нам не нужны эти дополнительные элементы, воспользуемся методом `Array.slice`, чтобы избавиться от них прежде, чем начать подсчет строк в наших файлах.

```
const fs = require('fs');

const filenames = process.argv.slice(2);
```

[2] Имя `argv` — это кивок языку C. *v* — это первая буква слова *vector* (вектор), который подобен массиву.

```
let counts = filenames.map(f => {
    try {
        const data = fs.readFileSync(f, { encoding: 'utf8' });
        return '${f}: ${data.split('\n').length}';
    } catch(err) {
        return '${f}: ошибка при чтении файла';
    }
});

console.log(counts.join('\n'));
```

Объект `process` позволяет также обращаться к переменным среды окружения через объект `process.env`. Переменные среды окружения называют системными переменными, которые главным образом используются для программ командной строки. В большинстве систем Unix вы можете установить переменную среды окружения, просто введя команду `export` *`ИМЯ=Значение`* (традиционно имена переменных окружения пишутся прописными буквами). В Windows для этого используется команда `set` *`ИМЯ=Значение`*. Переменные окружения зачастую используются для изменения поведения некого аспекта вашей программы, когда нежелательно передавать значение в командной строке каждый раз при запуске программы.

Например, мы могли бы использовать среду окружения для отключения вывода отладочной информации. Для управления поведением программы используется переменная среды окружения `DEBUG`, которую мы устанавливаем равной `1`, если хотим, чтобы программа выводила отладочную информацию (любое другое значение отключит отладку).

```
const debug = process.env.DEBUG === "1" ?
    console.log :
    function() {};

debug("Выводится в случае, если переменная окружения DEBUG=1!");
```

В этом примере мы создаем функцию, `debug`, которая просто является псевдонимом для `console.log`, если переменная окружения `DEBUG` установлена, и *пустой функцией* (которая не делает ничего) в противном случае (если бы мы оставили переменную `debug` неопределенной, то вызвали бы ошибку, когда попытались бы использовать ее!).

В предыдущем разделе мы говорили о текущем рабочем каталоге, которым по умолчанию является каталог запуска программы (а не каталог, в котором программа расположена). Метод `process.cwd` указывает текущий рабочий каталог, а `process.chdir` позволяет его изменить. Например, если нужно вывести каталог, из которого программа была запущена, и изменить текущий каталог на тот каталог, в котором находится сама программа, то можно сделать следующее.

```
console.log('Текущий каталог: ${process.cwd()}');
process.chdir(__dirname);
console.log('Новый текущий каталог: ${process.cwd()}');
```

Информация об операционной системе

Модуль `os` предоставляет некую специфическую для платформы информацию о компьютере, на котором выполняется приложение. Вот пример, демонстрирующий самую полезную информацию, предоставляемую модулем `os` и их значения, полученные на моем компьютере.

```
const os = require('os');

console.log("Имя хоста: " + os.hostname());          // prometheus
console.log("Тип ОС: " + os.type());                 // Linux
console.log("Платформа: " + os.platform());          // linux
console.log("Версия: " + os.release());              // 3.13.0-52-generic
console.log("Время работы: " +
    (os.uptime()/60/60/24).toFixed(1) + " days");    // 80.3 days
console.log("Архитектура процессора: " + os.arch()); // x64
console.log("Количество процессоров: " + os.cpus().length); // 1
console.log("Объем памяти: " +
    (os.totalmem()/1e6).toFixed(1) + " MB");         // 1042.3 MB
console.log("Свободно: " +
    (os.freemem()/1e6).toFixed(1) + " MB");          // 195.8 MB
```

Дочерние процессы

Модуль `child_process` позволяет вашему приложению запускать другие программы, будь то другие программы Node, а также исполняемые файлы или сценарии на другом языке. Описание всех подробностей управления дочерними процессами выходит за рамки этой книги, но простой пример мы рассмотрим.

Модуль `child_process` предоставляет три основные функции: `exec`, `execFile` и `fork`. Как и у модуля `fs`, здесь есть синхронные версии этих функций (`execSync`, `execFileSync` и `forkSync`). Функции `exec` и `execFile` могут запустить любой выполняемый файл, поддерживаемый вашей операционной системой. Функция `exec` вызывает *оболочку* (это то, что лежит в основе командной строки вашей операционной системы; если вы можете запустить нечто из командной строки, вы можете запустить это с помощью функции `exec`). Функция `execFile` позволяет запустить исполняемый файл непосредственно; она обеспечивает немного улучшенное использование памяти и ресурсов, но требует большего внимания. Наконец функция `fork` позволяет запускать другие сценарии Node (что также может быть сделано функцией `exec`).

Функция `fork` запускает отдельный процессор Node, поэтому расход ресурсов будет таким же, как и при использовании функции `exec`; но функция `fork` позволяет обращаться к некоторым возможностям *взаимодействия между процессами* (interprocess communication). Более

подробная информация по этой теме приведена в *официальной документации* (https://nodejs.org/api/child_process.html#child_process_child_process_fork_modulepath_args_options).

Поскольку функция exec является наиболее общей и наименее требовательной, в этой главе мы будем использовать ее.

В демонстрационных целях мы выполним команду dir, которая отображает список содержимого каталога (хотя пользователям Unix более знакома команда ls, в большинстве систем Unix она равнозначна команде dir).

```
const exec = require('child_process').exec;

exec('dir', function(err, stdout, stderr) {
    if(err) return console.error('Ошибка при запуске "dir"');
    stdout = stdout.toString(); // преобразует Buffer в строку
    console.log(stdout);
    stderr = stderr.toString();
    if(stderr !== '') {
        console.error('Ошибка:');
        console.error(stderr);
    }
});
```

Поскольку функция exec запускает системную оболочку, мы не должны указывать путь к каталогу, где хранится выполняемый файл. Чтобы вызвать определенную программу, которая обычно недоступна из оболочки вашей системы, необходимо указать полный путь к ее исполняемому файлу.

Функции обратного вызова метода exec помимо признака ошибки передается два объекта типа Buffer — один для stdout (стандартный поток вывода программы) и другой для stderr (стандартный поток вывода ошибок, если они есть). В этом примере, поскольку мы не предусматриваем вывод на устройство stderr, мы сначала проверяем значение первого аргумента на предмет возникновения ошибки в процессе запуска программы, а затем выводим полученные результаты на консоль.

Функции exec можно передать опциональный объект options, который позволяет задать рабочий каталог, переменные окружения и т.д. Более подробная информация по этой теме приведена в *официальной документации* (https://nodejs.org/api/child_process.html).

Обратите внимание на способ, которым мы импортируем функцию exec. Вместо того чтобы импортировать модуль child_process, используя const child_process = require('child_process'), а затем вызывать метод exec как child_process.exec, мы непосредственно используем псевдоним exec. Вы можете пользоваться любым вариантом, но тот способ, которым мы это сделали, весьма распространен.

Потоки

Концепция *потока* (stream) важна для Node. Поток — это объект, который имеет дело с данными (как и подразумевает его название) в потоке (слово *поток* должно заставить вас думать о *течении*, а поскольку течение — это нечто, происходящее во времени, ему имеет смысл быть асинхронным).

Потоки могут быть *потоками чтения*, *записи* или того и другого (*дуплексные потоки*). Потоки имеют смысл тогда, когда передача данных осуществляется на протяжении некоторого времени. Примерами могут служить ввод пользователем данных с клавиатуры или веб-службы с двусторонней связью с клиентом. При доступе к файлам также зачастую используются потоки (даже при том, что мы вполне можем читать и писать в файлы без потоков). Мы будем использовать файловые потоки для демонстрации создания потоков, чтения из них и записи, а также создания *канала* (pipe) между ними.

Начнем с создания потока записи и запишем в него данные.

```
const fs = require('fs');

const ws = fs.createWriteStream('stream.txt', { encoding: 'utf8' });
ws.write('Строка 1\n');
ws.write('Строка 2\n');
ws.end();
```

Методу `end` можно дополнительно передать аргумент данных, тогда он будет эквивалентен вызову метода `write`. Таким образом, если нужно вывести данные только один раз, можете просто вызвать метод `end` с теми данными, которые вы хотите записать.

В наш поток записи (write stream — `ws`) можно выводить данные с помощью метода `write` до вызова метода `end`, после чего поток будет закрыт, и дальнейшие вызовы метода `write` приведут к ошибке. Поскольку вы можете вызывать метод `write` столько раз, сколько необходимо, а затем вызвать метод `end`, поток записи идеален для записи данных в течение некоторого времени.

Точно так же мы можем создать поток чтения, чтобы читать данные по мере их поступления.

```
const fs = require('fs');

const rs = fs.createReadStream('stream.txt', { encoding: 'utf8' });
rs.on('data', function(data) {
    console.log('>> Данные: ' + data.replace('\n', '\\n'));
});
rs.on('end', function(data) {
    console.log('>> Конец');
});
```

В этом примере мы просто выводим содержимое файла на консоль (заменяя символы перехода на новую строку для наглядности). Вы можете поместить оба эти примера в один и тот же файл: у вас может быть поток записи, пишущий в файл, и поток чтения, читающий из него.

Дуплексные потоки не столь распространены и не рассматриваются в этой книге. Как и следовало ожидать, вы можете вызвать метод `write`, чтобы писать данные в дуплексный поток, а также прослушивать события `data` и `end`.

Поскольку данные в потоках "текут", вполне резонно взять данные, выходящие из потока чтения, и перенаправить их в поток записи. Этот процесс называется *конвейером* (piping). Например, мы могли бы перенаправить поток чтения в поток записи, чтобы скопировать содержимое одного файла в другой.

```
const rs = fs.createReadStream('stream.txt');
const ws = fs.createWriteStream('stream_copy.txt');
rs.pipe(ws);
```

Обратите внимание, что в этом примере мы не должны определять кодировку символов: `rs` просто пересылает байты из файла `stream.txt` в поток `ws` (что приводит к их записи в файл `stream_copy.txt`); кодировка символов имеет значение, только если мы пытаемся интерпретировать данные.

Конвейерная обработка — это общая методика для перемещения данных. Например, можно переслать содержимое файла в виде ответа веб-сервера. Либо вы могли бы переслать сжатые данные процессору распаковки, который, в свою очередь, перешлет данные программе записи в файл.

Веб-серверы

Хотя Node теперь используется во многих приложениях, его первоначальная цель состояла в предоставлении услуг веб-сервера. Таким образом, нельзя не рассмотреть и этот способ его применения.

Те из вас, кто настраивал сервер Apache (или IIS, или любой другой веб-сервер), могут быть поражены простой создания и функционирования этого веб-сервера. Модуль `http` (и его защищенный дубликат, модуль `https`) предоставляет метод `createServer`, который создает простой веб-сервер. Все, что вы должны сделать, — это указать функцию обратного вызова, которая будет обрабатывать входящие запросы. Чтобы запустить сервер, нужно просто вызывать его метод `listen` и указать номер прослушиваемого порта.

```
const http = require('http');

const server = http.createServer(function(req, res) {
    console.log('${req.method} ${req.url}');
    res.end('Привет, мир!');
```

```
});

const port = 8080;
server.listen(port, function() {
    // методу listen передается функция обратного вызова,
    // которая вызывается после запуска сервера
console.log('Сервер запущен на порту ${port}');
});
```

Из соображений безопасности в большинстве операционных систем запрещено прослушивать стандартный порт HTTP (80) без запроса на повышение прав. Фактически повышенные права необходимы для прослушивания любого порта ниже 1024. Разумеется, это сделать не сложно: если у вас есть доступ к команде `sudo`, можете запустить свой сервер через `sudo` и, получив права администратора, начать прослушивать порт 80. Для целей разработки и отладки обычно используются порты выше 1024. Обычно выбирают такие номера, как 3000, 8000, 3030 и 8080, поскольку их легче запомнить.

Если вы запустите эту программу и перейдете в браузере по адресу `http://localhost:8080`, то увидите строку `Привет, мир!`. На консоли мы регистрируем все запросы, которые состоят из метода и пути URL. Вас может удивить тот факт, что каждый раз при переходите в браузере по этому URL, на сервер отправляется два запроса.

```
GET /
GET /favicon.ico
```

Большинство браузеров неявно запрашивают пиктограмму, которую они затем отображают на панели URL или заголовке вкладки. Поэтому мы видим этот запрос на нашей консоли.

В основе веб-сервера Node лежит функция обратного вызова, которую вы должны указать при создании сервера. Именно она обрабатывает все входящие запросы. Ей передается два аргумента, объект `IncomingMessage` (зачастую для него выбирается переменная `req`) и объект `ServerRequest` (зачастую для него выбирается переменная `res`). Объект `IncomingMessage` содержит всю информацию о HTTP-запросе: какой URL затребован, все посланные заголовки, все посланные в теле данные и т.д. Объект `ServerResponse` содержит свойства и методы для управления ответом, который отсылается назад клиенту (обычно браузеру). Если вы увидели, что мы вызвали метод `req.end`, и задались вопросом "Является ли `req` потоком записи?", то просмотрите заголовок класса. Объект `ServerResponse` реализует интерфейс потока записи, который определяет то, как именно данные пересылаются клиенту. Поскольку объект `ServerResponse` — это поток записи, он облегчает передачу файла... но нам ничто не мешает создать поток для чтения файла и переслать его в качестве ответа

HTTP-сервера. Например, если у вас есть файл `favicon.ico`, улучшающий внешний вид вашего веб-сайта, вы можете выделить этот запрос и отправить содержимое данного файла непосредственно клиенту.

```
const server = http.createServer(function(req, res) {
    if(req.method === 'GET' && req.url === '/favicon.ico') {
        const fs = require('fs');
        fs.createReadStream('favicon.ico');
        fs.pipe(res);              // это вместо вызова метода 'end'
    } else {
        console.log('${req.method} ${req.url}');
        res.end('Hello world!');
    }
});
```

Выше приведен минимально возможный, хотя и не очень интересный, веб-сервер. Анализируя информацию, содержащуюся в объекте `IncomingRequest`, вы можете расширить приведенную выше модель и создать любой вид веб-сайта по своему желанию.

Если вы планируете использовать Node для обслуживания веб-сайта, то вам, вероятно, понадобится изучить использование таких каркасов, как *Express* или *Koa*, которые возьмут на себя часть работы по построению веб-сервера с нуля.

Koa — это преемник весьма популярного каркаса Express, и это не случайно: оба написаны Ти Джей Головайчуком. Если вы уже знакомы с Express, то и с Koa вы почувствуете себя как дома, за исключением только того, что в нем применяется подход к веб-разработке, более ориентированный на ES6.

Заключение

Здесь мы поверхностно затронули самые важные моменты интерфейса API Node. Мы сосредоточились на тех пакетах, которые вы, вероятно, увидите почти в каждом приложении (таких, как `fs`, `Buffer`, `process` и `stream`). Однако существует и множество других пакетов, которые вы должны будете изучить самостоятельно. *Официальная документация* (`https://nodejs.org/en/docs/`) — очень подробна, но для новичка может быть сложной. Если вас интересует разработка приложений для Node, рекомендую начать с книги Шелли Пауэрса (Shelley Powers) *Learning Node*.

Свойства объекта и прокси-объекты

Свойства доступа: получатели и установщики

Существует два типа свойств объектов: *свойства данных* (data property) и *свойства доступа* (accessor property). Мы уже сталкивались с обоими типами, но свойства доступа остались за кадром благодаря некоторым синтаксическим нововведениям ES6 (в главе 9 мы называли их "динамическими свойствами").

Мы знакомы с функциональными свойствами (или методами); свойства доступа подобны им, но у них есть *две* функции (*получения значения* (getter) и *установки значения* (setter)), которые при доступе к ним действуют скорее как свойство данных, чем как функция.

Давайте рассмотрим динамические свойства. Предположим в классе `User` есть методы `setEmail` и `getEmail`. Мы решили использовать методы "get" и "set" вместо обычного свойства `email` потому, что хотим предотвратить ввод пользователем недопустимого адреса электронной почты. Наш класс очень прост (для простоты мы считаем любую строку, содержащую символ "`@`" допустимым адресом электронной почты).

```
const USER_EMAIL = Symbol();
class User {
    setEmail(value) {
        if(!/@/.test(value)) throw new Error('Неправильный адрес: ${value}');
        this[USER_EMAIL] = value;
    }
    getEmail() {
        return this[USER_EMAIL];
    }
}
```

Единственное, что в этом примере заставляет нас использовать два метода (вместо обычного свойства данных), — это предотвращение присваивания свойству USER_EMAIL недопустимого адреса электронной почты. Здесь мы используем символьное свойство, чтобы блокировать случайный прямой доступ к свойству данных, содержащему адрес электронной почты. Если бы мы назвали строковое свойство email или даже _email, то было бы довольно просто по небрежности обратиться к нему непосредственно.

Это типичный сценарий использования, и он прекрасно работает, но несколько громоздким, чем нам хотелось бы. Вот пример использования этого класса.

```
const u = new User();
u.setEmail("john@doe.com");
console.log('Адрес пользователя: ${u.getEmail()}');
```

Хотя это вполне сработает, было бы естественнее написать

```
const u = new User();
u.email = "john@doe.com";
console.log('Адрес пользователя: ${u.email}');
```

Введем свойства доступа: они позволят сохранить преимущества прежнего подхода с естественным синтаксисом последнего. Давайте перепишем наш класс, используя свойства доступа.

```
const USER_EMAIL = Symbol();
class User {
    set email(value) {
        if(!/@/.test(value)) throw new Error('Неправильный адрес: ${value}');
        this[USER_EMAIL] = value;
    }
    get email() {
        return this[USER_EMAIL];
    }
}
```

Мы создали две разные функции, но они связаны с единым свойством email. Если значение свойству присваивается, то вызывается *функция установки значения* (с передачей присваиваемого значения в качестве первого аргумента), а если значение свойства запрашивается, то вызывается *функция получения значения*.

Можно создать функцию-получатель без функции-установщика значения; рассмотрим, например, функцию-получатель, которая возвращает периметр прямоугольника.

```
class Rectangle {
    constructor(width, height) {
        this.width = width;
        this.height = height;
```

```
    }
    get perimeter() {
        return this.width*2 + this.height*2;
    }
}
```

Здесь нет функции-установщика значения для периметра потому, что нет никакого очевидного способа определить ширину и высоту прямоугольника исходя из длины периметра; это свойство имеет смысл сделать свойством только для чтения.

Аналогично вы можете создать функцию-установщик без функции-получателя, хотя это намного менее распространенный вариант.

Атрибуты свойств объекта

К настоящему моменту вы уже достаточно поработали со свойствами объектов. Известно, что у них есть ключ (который может быть строкой или символом) и значение (которое может иметь любой тип). Мы также знаем, что порядок следования свойств в объекте не гарантируется (как, например, у массива или объекта `Map`). Известны два способа обращения к свойствам объекта (доступ к члену с использованием точечной формы записи и вычисляемый доступ к члену с использованием квадратных скобок). Наконец известны три способа создания свойства с помощью литеральной формы записи объекта (обычные свойства с ключами, которые являются идентификаторами, вычисляемые имена свойств, позволяющие обойтись без идентификаторов и использовать символы, а также сокращения методов).

Как бы то ни было, о свойствах необходимо знать больше. В частности, у свойств есть *атрибуты*, контролирующие поведение свойства в контексте объекта, которому они принадлежат. Начнем с создания свойства, используя одну из известных методик, а затем используем метод `Object.getOwnPropertyDescriptor` для исследования его атрибутов.

```
const obj = { foo: "bar" };
Object.getOwnPropertyDescriptor(obj, 'foo');
```

Это код возвратит следующее.

```
{ value: "bar", writable: true, enumerable: true, configurable: true }
```

Термины *атрибут свойства* (property attribute), *дескриптор свойства* (property descriptor) и *конфигурация свойства* (property configuration) используются как синонимы; все они означают одно и то же.

Мы увидели три разных атрибута свойства.

- Атрибут `writable` (перезаписываемый) определяет, может ли значение свойства быть изменено.

- Атрибут `enumerable` (перечислимый) определяет, будет ли свойство участвовать в перечислении свойств объекта (с использованием `for...in`, `Object.keys` или оператора расширения).
- Атрибут `configurable` (перестраиваемый) определяет, может ли свойство быть удалено из объекта или могут ли быть изменены его атрибуты.

Мы можем управлять атрибутами свойства, используя метод `Object.defineProperty`. Он позволяет создавать новые свойства или изменять существующие (если это свойство перестраиваемое).

Например, чтобы сделать свойство `foo` объекта `obj` доступным только для чтения, можно использовать `Object.defineProperty` так.

```
Object.defineProperty(obj, 'foo', { writable: false });
```

Теперь, если мы попытаемся присвоить значение свойству `foo`, то получим ошибку.

```
obj.foo = 3;
//
TypeError: Нельзя присваивать значение свойству 'foo', доступному только чтение
```

Попытка изменить значение свойства только для чтения закончится ошибкой только в строгом режиме. В нестрогом режиме ничего присвоено не будет, но и ошибки при этом тоже не будет.

Мы можем также использовать метод `Object.defineProperty` для добавления к объекту нового свойства. Это особенно полезно для свойств с атрибутами, поскольку в отличие от свойств данных нет никакого другого способа добавить свойство доступа после того, как объект был создан. Давайте добавим к объекту `obj` свойство `color` (на сей раз мы не будем заботиться о символах или проверке правильности).

```
Object.defineProperty(obj, 'color', {
    get: function() { return this.color; },
    set: function(value) { this.color = value; },
});
```

Чтобы создать свойство данных, нужно указать его *значение* в параметрах при вызове `Object.defineProperty`. Добавим к объекту `obj` свойства `name` и `greet`.

```
Object.defineProperty(obj, 'name', {
    value: 'Cynthia',
});
Object.defineProperty(obj, 'greet', {
    value: function() { return 'Привет, меня зовут ${this.name}!'; }
});
```

Один из популярных случаев применения `Object.defineProperty` — это сделать свойства неперечислимыми в массиве. Мы упоминали прежде, что не стоит

использовать строковое или символьное свойство в массиве, поскольку это противоречит самой идее применения массива, но это может быть полезно, если сделано осторожно и осмысленно. Хотя применение цикла `for...in` или `Object.keys` для массива также не очень хорошо (вместо них рекомендуется использовать `for`, `for...of` или `Array.prototype.forEach`), вы не можете запретить людям ими пользоваться. Поэтому, добавляя нечисловые свойства к массиву, необходимо делать их неперечислимыми на случай, если кто-то (по неосторожности) воспользуется массивом `for...in` или `Object.keys`. Вот пример добавления к массиву методов `sum` и `avg`.

```
const arr = [3, 1.5, 9, 2, 5.2];
arr.sum = function() { return this.reduce((a, x) => a+x); }
arr.avg = function() { return this.sum()/this.length; }
Object.defineProperty(arr, 'sum', { enumerable: false });
Object.defineProperty(arr, 'avg', { enumerable: false });
```

Мы могли бы также сделать это за один этап для каждого свойства.

```
const arr = [3, 1.5, 9, 2, 5.2];
Object.defineProperty(arr, 'sum', {
    value: function() { return this.reduce((a, x) => a+x); },
    enumerable: false
});
Object.defineProperty(arr, 'avg', {
    value: function() { return this.sum()/this.length; },
    enumerable: false
});
```

Наконец есть также метод `Object.defineProperties` (обратите внимание на название во множественном числе!), который позволяет определить сразу несколько свойств для объекта. Таким образом, мы можем переписать предыдущий пример как

```
const arr = [3, 1.5, 9, 2, 5.2];
Object.defineProperties(arr,
    sum: {
        value: function() { return this.reduce((a, x) => a+x); },
        enumerable: false
    }),
    avg: {
        value: function() { return this.sum()/this.length; },
        enumerable: false
    })
);
```

Защита объектов: замораживание, запечатывание и запрет расширения

Гибкая природа языка JavaScript позволяет создавать очень мощный код, однако одновременно она является и причиной всех проблем. Поскольку любой код в любом

месте может изменить объект любым способом, довольно просто написать код, который непреднамеренно или, что еще хуже, преднамеренно, будет делать опасные вещи.

В JavaScript предусмотрено три механизма для предотвращения неумышленных изменений (и затруднения злонамеренных): *замораживание* (freezing), *запечатывание* (sealing) и *запрет расширения* (preventing extension).

Замораживание предотвращает *любые* изменения объекта. Как только вы замораживаете объект, вы не можете

- установить значение его свойств;
- вызывать методы, которые изменяют значение свойств объекта;
- вызывать функции установки значения объекта (которые изменяют значение свойств объекта);
- добавлять новые свойства;
- добавлять новые методы;
- изменять конфигурацию существующих свойств или методов.

По сути, замораживание объекта делает его неизменным. Это полезно для объектов-данных, поскольку замораживание объекта, содержащего методы делает бесполезным любые методы, которые изменяют состояние объекта.

Чтобы заморозить объект, используйте метод `Object.freeze` (чтобы выяснить, заморожен ли объект, вызовите метод `Object.isFrozen`). Предположим, например, что у вас есть объект, который вы используете для хранения неизменяемой информации о своей программе (такой, как название компании, версия, идентификатор сборки и метод получения информации об авторских правах).

```
const appInfo = {
    company: 'White Knight Software, Inc.',
    version: '1.3.5',
    buildId: '0a995448-ead4-4a8b-b050-9c9083279ea2',
    // эта функция только читает значения свойств, поэтому замораживание
    // на нее не повлияет
    copyright() {
        return 'c ${new Date().getFullYear()}, ${this.company}';
    },
};
Object.freeze(appInfo);
Object.isFrozen(appInfo); // true

appInfo.newProp = 'test';
// TypeError: Нельзя добавить свойство newProp, объект не расширяем

delete appInfo.company;
// TypeError: Нельзя удалить свойство 'company'
```

```
appInfo.company = 'test';
// TypeError: Нельзя присваивать значение свойству только для чтения 'company'

Object.defineProperty(appInfo, 'company', { enumerable: false });
// TypeError: Нельзя переопределять свойство: company
```

Запечатывание объекта предотвращает добавление новых свойств, реконфигурацию или удаление существующих свойств. Запечатывание применяется, когда нужно, чтобы работали все методы экземпляра класса, изменяющие свойства объекта (по крайней мере до тех пор, пока они не попытаются перенастроить эти свойства). Вы можете запечатать объект методом `Object.seal`, а узнать, запечатан ли объект, — вызвав метод `Object.isSealed`:

```
class Logger {
    constructor(name) {
        this.name = name;
        this.log = [];
    }
    add(entry) {
        this.log.push({
            log: entry,
            timestamp: Date.now(),
        });
    }
}

const log = new Logger("Бортовой журнал");
Object.seal(log);
Object.isSealed(log);                    // true

log.name = "Бортовой журнал капитана";        // OK
log.add("Еще один скучный день на море...."); // OK

log.newProp = 'test';
// TypeError: Нельзя добавить свойство newProp, объект не расширяем

log.name = 'test';                       // OK

delete log.name;
// TypeError: Нельзя удалить свойство 'name'

Object.defineProperty(log, 'log', { enumerable: false });
// TypeError: Нельзя переопределить свойство: log
```

И наконец самая слабая защита — сделать объект нерасширяемым, что запретит только добавление новых свойств. Свойствам могут быть присвоены значения, они могут быть удалены и реконфигурированы. Мы можем продемонстрировать методы

`Object.preventExtensions` и `Object.isExtensible` на примере нашего прежнего класса `Logger`.

```
const log2 = new Logger("Журнал первого помощника");
Object.preventExtensions(log2);
Object.isExtensible(log2); // true

log2.name = "Бортовой журнал первого помощника";     // OK
log2.add("Еще один скучный день на море...."); // OK

log2.newProp = 'test';
// TypeError: Нельзя добавить свойство newProp, объект не расширяем

log2.name = 'test';                  // OK
delete log2.name;                    // OK
Object.defineProperty(log2, 'log',
{ enumerable: false });              // OK
```

Я использую метод `Object.preventExtensions` не очень часто. Для предотвращения модификации объекта обычно я также хочу предотвратить удаление и реконфигурацию его свойств, поэтому я предпочитаю запечатывать объект.

Возможности защиты объектов приведены в табл. 21.1.

Таблица 21.1. Возможности защиты объектов

Действие	Обычный объект	Замороженный объект	Запечатанный объект	Нерасширяемый объект
Добавление свойства	Разрешено	Запрещено	Запрещено	Запрещено
Чтение свойства	Разрешено	Разрешено	Разрешено	Разрешено
Установка значения свойства	Разрешено	Запрещено	Разрешено	Разрешено
Перенастройка свойства	Разрешено	Запрещено	Запрещено	Разрешено
Удаление свойства	Разрешено	Запрещено	Запрещено	Разрешено

Прокси-объекты

Прокси-объекты (proxy) — это нововведение ES6, которое обеспечивает дополнительные функциональные возможности *метапрограммирования* (metaprogramming), т.е. способности программы изменять саму себя.

Прокси-объект, по существу, способен перехватывать и (опционально) изменять действия объекта. Для начала рассмотрим простой пример: изменение свойства доступа. Начнем с обычного объекта, у которого есть несколько свойств.

```
const coefficients = {
    a: 1,
    b: 2,
    c: 5,
};
```

Предположим, что свойства этого объекта представляют коэффициенты в математическом уравнении. Мы могли бы использовать его так.

```
function evaluate(x, c) {
    return c.a + c.b * x + c.c * Math.pow(x, 2);
}
```

Пока неплохо... Теперь мы можем хранить коэффициенты квадратного уравнения в объекте и вычислять уравнение для любого значения x. Но что если мы передадим объект с недостающими коэффициентами?

```
const coefficients = {
    a: 1,
    c: 3,
};
evaluate(5, coefficients); // NaN
```

Мы могли бы решить проблему, установив `coefficients.b` равным `0`, но прокси-объекты предоставляют нам лучшую возможность. Поскольку они способны перехватывать действия с объектом, мы можем гарантировать, что неопределенные свойства всегда будут иметь значение `0`. Давайте создадим прокси-объект для нашего объекта `coefficients`.

```
const betterCoefficients = new Proxy(coefficients, {
    get(target, key) {
        return target[key] || 0;
    },
});
```

На момент написания этой книги прокси-объекты не поддерживались в Babel. Однако они поддерживаются в текущем выпуске Firefox, и эти примеры кода можно проверить там.

Первый аргумент конструктора `Proxy` — *целевой объект* (target), или объект, к которому применяется прокси-объект. Второй аргумент — *обработчик* (handler), который определяет перехватываемые действия. В данном случае мы перехватываем только доступ к свойствам, обозначаемый функцией `get`. Этот процесс несколько отличается от методов доступа к свойствам `get...`, поскольку он работает *и* для обычных свойств, *и* для методов доступа. Функции `get` передается три аргумента (мы используем только первые два): целевой объект, ключ свойства (строка или символ) и получатель (сам прокси-объект или нечто происходящее от него).

В этом примере мы просто выясняем, установлен ли ключ на целевом объекте, и если не установлен, то возвращаем значение `0`. Давайте опробуем это.

```
betterCoefficients.a;        // 1
betterCoefficients.b;        // 0
betterCoefficients.c;        // 3
```

```
betterCoefficients.d;         // 0
betterCoefficients.anything; // 0;
```

По существу, мы создали прокси-объект для нашего объекта `coefficients`, который способен иметь бесконечное количество свойств (все устанавливаются равными `0`, кроме тех, которые были определены явно)!

Мы могли бы еще несколько улучшить свой прокси-объект, чтобы он обрабатывал только свойства, имена которых состоят из одиночных строчных букв.

```
const betterCoefficients = new Proxy(coefficients, {
    get(target, key) {
        if(!/^[a-z]$/.test(key)) return target[key];
        return target[key] || 0;
    },
});
```

Вместо простой проверки на существование свойства `target[key]` мы могли бы возвращать `0`, если его значение не является числом... Я оставляю это упражнение для читателя.

Точно так же мы можем перехватить свойства (или методы доступа), устанавливаемые обработчиком `set`. Давайте рассмотрим пример, в котором у объекта имеются опасные свойства. Мы хотим воспрепятствовать установке значений этих свойств и вызову методов-установщиков без дополнительного этапа контроля. Дополнительным этапом контроля, который мы будем использовать перед обращением к опасным функциям, является установка свойства `allowDangerousOperations` равным значению `true`.

```
const cook = {
    name: "Walt",
    redPhosphorus: 100, // опасно
    water: 500,         // безопасно
};
const protectedCook = new Proxy(cook, {
    set(target, key, value) {
        if(key === 'redPhosphorus') {
            if(target.allowDangerousOperations)
                return target.redPhosphorus = value;
            else
                return console.log("Очень опасно!");
        }
        // все остальные свойства безопасны
        target[key] = value;
    },
});

protectedCook.water = 550;          // 550
protectedCook.redPhosphorus = 150; // Очень опасно!
```

```
protectedCook.allowDangerousOperations = true;
protectedCook.redPhosphorus = 150; // 150
```

В этом разделе мы весьма поверхностно рассмотрели основные функции прокси-объектов. Чтобы узнать больше, я рекомендую начать со статьи Акселя Роушмайера (Axel Rauschmayer) *Meta Programming with ECMAScript 6 Proxies* (`http://www.2ality.com/2014/12/es6-proxies.html`), а затем читать *документацию MDN* (https://developer.mozilla.org/ru/docs/Web/JavaScript/Reference/Global_Objects/Proxy).

Заключение

В этой главе мы приподняли занавес, который скрывает механизм объектов JavaScript, и получили подробную картину работы свойств объекта, а также способов изменения их поведения. Мы также узнали, как защитить объекты от изменения.

Наконец мы узнали о чрезвычайно полезной новой концепции ES6 — прокси-объектах. Прокси-объекты предоставляют мощные методики метапрограммирования, и я подозреваю, что мы еще увидим некоторые весьма интересные способы их использования в связи с ростом популярности преимуществ ES6.

Дополнительные ресурсы

Мое мнение о том, что JavaScript — это выразительный и мощный язык, сформировалось довольно давно. Это не "игрушечный" язык, который легко изучить или отбросить как "для начинающих". Вы наверняка это уже очень хорошо понимаете, потому что изучили предыдущие главы данной книги!

Моя задача в этой книге не в том, чтобы исчерпывающе описать каждое средство языка JavaScript, а в том, чтобы рассмотреть каждую наиболее важную методику программирования. Если JavaScript — это ваш первый язык, то вы только вначале своего пути. Я надеюсь, что дал вам стройную структуру информации, основываясь на которой, вы сможете стать экспертом.

Большая часть материала этой главы взята из моей первой книги — *Web Development with Node and Express* (издательство O'Reilly).

Сетевая документация

Для JavaScript, CSS и HTML документация *сети разработчиков Mozilla* (Mozilla Developer Network — MDN, `https://developer.mozilla.org/ru/`) не имеет равных. Если мне нужна документация по JavaScript, я либо ищу ее непосредственно в MDN, либо добавляю "mdn" к своему поисковому запросу. В противном случае в результатах поиска будет неизбежно присутствовать w3schools. Кто бы ни предоставлял услуги SEO для w3schools, он гений, но я рекомендую избегать этого сайта; я нахожу, что документации на нем зачастую недостает.

Хотя MDN — это великолепный справочник по HTML, если вы новичок в HTML5 (или даже не знакомы с ним), вам стоит прочитать книгу Эда Титтеля и Криса Минника HTML5 и CSS3 для чайников. Сообщество WHATWG поддерживает постоянно обновляемую *спецификацию HTML5* (`https://developers.whatwg.org/`); именно к нему я обычно обращаюсь в первую очередь в поисках ответов на действительно сложные вопросы по HTML. И наконец есть официальные спецификации HTML и CSS, расположенные на *веб-сайте W3C*; это сухие, трудные для чтения документы, но иногда это единственная надежда при решении самых серьезных проблем.

ES6 соответствует языковой спецификации *ECMA-262 ECMAScript 2015 Language Specification* (`http://www.ecma-international.org/ecma-262/6.0/`). Для проверки доступности средств ES6 в Node (и в различных браузерах) обращайтесь к превосходному руководству, поддерживаемому *@kangax* (`http://kangax.github.io/compat-table/es6/`).

Для *jQuery* (`http://api.jquery.com/`) и *Bootstrap* (`http://getbootstrap.com/`) есть чрезвычайно хорошая сетевая документация.

Документация по Node (`https://nodejs.org/api/`) является весьма высококачественной и исчерпывающей, и это должен быть ваш первый выбор при поиске авторитетной документации о модулях Node (таких, как `http`, `http` и `fs`). *Документация по npm* (`https://docs.npmjs.com/`) является исчерпывающей и полезной, особенно *страница о файле* `package.json` (https://docs.npmjs.com/getting-started/using-a-package.json).

Периодические издания

Есть три бесплатных периодических издания, на которые обязательно нужно подписаться и которые рекомендуется читать каждую неделю.

- *JavaScript Weekly* (`http://javascriptweekly.com/`)
- *Node Weekly* (`http://nodeweekly.com/`)
- HTML5 Weekly (`http://frontendfocus.co/`)

Эти издания позволяют быть в курсе последних новостей, служб, блогов и учебных курсов по мере того как они будут становиться доступными.

Блоги и учебные курсы

Блоги — это отличное средство быть в курсе событий по JavaScript. Но при чтении некоторых из этих блогов я не совсем был абсолютно согласен.

- В *блоге Акселя Роушмайера* (Axel Rauschmayer, `http://www.2ality.com/`) есть великолепные статьи о ES6 и связанных с ним технологиях. Д-р Роушмайер подходит к JavaScript с академической точки зрения информатики, но его статьи весьма доступны и просты для чтения, а специалисты в информатике оценят по достоинству дополнительные подробности, которые он предоставляет.
- В *блоге Нолана Лоусона* (Nolan Lawson, `https://nolanlawson.com/`) есть много прекрасных подробных постов о реальной разработке приложений на JavaScript. Его статья *We Have a Problem with Promises* (`https://pouchdb.com/2015/05/18/we-have-a-problem-with-promises.html`) обязательна для изучения.

- В *блоге Дэвида Уолша* (David Walsh, `https://davidwalsh.name/`) есть фантастические статьи о разработке приложений на JavaScript и о связанных с ним технологиях. Если вы испытывали затруднения при изучении главы 14, обязательно прочитайте его статью *The Basics of ES6 Generators* (`https://davidwalsh.name/es6-generators`).
- Блог `@kangax`, *Perfection Kills* (`http://perfectionkills.com/`), полон фантастических учебных пособий, упражнений и вопросов. Он настоятельно рекомендуется и для новичков, и для экспертов.

Теперь, когда вы прочитали эту книгу, доступные сетевые учебные пособия и упражнения должны быть для вас очень простыми. Но если вы все еще чувствуете, что пропустили некоторые фундаментальные части, или просто хотите попрактиковаться в основных принципах, я рекомендую следующее.

- Курсы *Lynda.com JavaScript* (`https://www.lynda.com/JavaScript-training-tutorials/244-0.html`).
- Курсы *Treehouse JavaScript* (`https://teamtreehouse.com/learn-to-code/javascript`).
- Курсы *Codecademy's JavaScript* (`https://www.codecademy.com/learn/javascript`).
- Вводный курс *Microsoft Virtual Academy's intro course on JavaScript* (`https://mva.microsoft.com/en-US/training-courses/javascript-fundamentals-for-absolute-beginners-14194`). Если вы пишете код JavaScript для систем Windows, я рекомендую это как материал по использованию Visual Studio для разработки приложений на JavaScript.

Система Stack Overflow

Весьма велики шансы, что вы уже используете популярную систему вопросов и ответов Stack Overflow (SO). Будучи запущенной в 2008 году, она стала доминирующим сетевым сайтом вопросов и ответов для программистов, и это ваш наилучший ресурс, чтобы задавать вопросы и получать ответы по JavaScript (и любой другой технологии, описанной в этой книге). Stack Overflow — это поддерживаемый сообществом сайт вопросов и ответов на основании репутации. Модель репутации — это то, что отвечает за качество сайта и его успех. Пользователи могут зарабатывать репутацию в ходе "голосования" по их вопросам или ответам или давая одобренные ответы. Чтобы задать вопрос, наличие репутации необязательно, а регистрация бесплатна. Однако есть вещи, которые вы можете сделать, чтобы увеличить вероятность получения ответа на свой вопрос, что мы и обсудим в этом разделе.

Репутация — это валюта Stack Overflow. И хотя там есть люди, которые искренне хотят вам помочь, это также дает им шанс заработать себе репутацию — достаточно большой соблазн, мотивирующий давать хорошие ответы. В сообществе SO есть много действительно умных людей, и все они конкурируют между собой, чтобы предоставить первый и/или лучший правильный ответ на ваш вопрос (к счастью, крайне не выгодно давать быстрые, но плохие ответы). Вот что вы можете сделать, чтобы увеличить возможности получения хорошего ответа на свой вопрос.

Будьте осведомлены

Пройдите обзорный тур *SO tour* (`http://stackoverflow.com/tour`), а затем прочитайте раздел *How do I ask a good question?* (Как задать хороший вопрос?, `http://stackoverflow.com/help/how-to-ask`). Если хотите, можете прочитать всю *вспомогательную документацию* (*help documentation*, `http://stackoverflow.com/help`) и заработать значок!

Не задавайте вопросы, на которые уже отвечали

Проявите усердие и попытайтесь поискать ответ; возможно, кто-то уже задал ваш вопрос. Если вы зададите вопрос, ответ на который легко найти в SO, то ваш вопрос будет быстро закрыт как повторный, и за это люди проголосуют против вас, что негативно скажется на вашей репутации.

Не просите людей писать ваш код вместо вас

Вы быстро обнаружите, что ваш вопрос проголосован "против" и закрыт, если просто спросите "Как мне сделать..." Сообщество SO ожидает, что вы прикладывали усилия для решения своей проблемы, прежде чем обратиться к SO. Опишите в своем вопросе, что вы пытались сделать и почему это не сработало.

Задавайте по одному вопросу за раз

Отвечать сразу на несколько вопросов ("Как я могу сделать это... а затем это... а затем что-то еще и как поступить лучше?") трудно; они отбивают желание отвечать.

Предоставьте краткий пример своей проблемы

Я ответил на множество вопросов SO, но я почти автоматически пропускаю вопросы, когда вижу в них три (или больше!) страницы кода. Просто взять свой файл из 5000 строк и вставить его в вопрос SO — это не наилучший способ добраться до ответа (но именно так люди все время и делают). Это "ленивый" подход, который не часто вознаграждается. Мало того что вы скорее всего не получите полезный ответ, сам процесс устранения элементов кода, которые явно не являются причиной проблемы, может привести вас к решению (и вам даже не понадобится задавать вопрос на SO). Создание краткого примера позволит вам приобрести навыки в отладке и научиться мыслить критически, что сделает вас хорошим гражданином SO.

Изучите язык разметки Markdown

Для оформления вопросов и ответов в Stack Overflow используется язык разметки Markdown. У хорошо оформленного вопроса больше шансов на ответ, поэтому имеет смысл потратить время на изучение этого полезного и все более и более вездесущего *языка разметки*.

Ответы и голосование

Если кто-то отвечает на ваш вопрос удовлетворительно, проголосуйте за него и примите его ответ; это повысит репутацию отвечающей стороны и SO. Если приемлемые ответы предоставляет несколько человек, выберите того, кого считаете лучшим, и примите его ответ, а также проголосуйте за всех, кто, по вашему мнению, предоставил полезный ответ.

Если вы решите свою проблему раньше остальных, ответьте на собственный вопрос

SO — это общественный ресурс; если у вас возникла проблема, возможно, она есть и у кого-то еще. Если вы самостоятельно решили ее, ответьте на собственный вопрос для общей пользы.

Если вам нравится помогать сообществу, попробуйте сами отвечать на вопросы: это интересно и полезно, а также может дать преимущества, куда более материальные, чем вероятный бал репутации. Если у вас есть вопрос, на который вы не получали полезных ответов в течение двух дней, можете назначить *премию* (bounty) за ответ, используя собственную репутацию. С вашей учетной записи будут немедленно сняты пункты репутации, и это невозмещаемо. Если кто-то отвечает на вопрос удовлетворительно и вы принимаете ответ, он получает премию. Конечно, для назначения премии у вас должна быть достаточная репутация: минимальная премия — 50 пунктов репутации. Хотя вы можете заработать репутацию, задавая качественные вопросы, как правило, можно быстрее заработать ее, предоставляя качественные ответы.

У ответов на вопросы есть также то преимущество, что это отличный способ самообучения. Отвечая на вопросы других, я обычно чувствую, что узнаю больше, чем когда ищу ответы на собственные вопросы. Если вы действительно хотите полностью изучить технологию, узнайте ее основы, а затем попытайтесь заняться вопросами других на SO. Поначалу вы, может быть, будете стесняться людей, которые уже являются экспертами, но со временем вы заметите, что сами стали одним из экспертов.

Наконец вы не должны смущаться, используя свою репутацию для продвижения своей карьеры. Хорошую репутацию, безусловно, нужно упоминать в резюме. Это сработало у меня, и теперь, когда я сам интервьюирую разработчиков, меня всегда впечатляет их хорошая репутация SO (хорошей я считаю репутацию SO более 3000; репутация с пятью знаками — *великолепна*). Хорошая репутация SO говорит мне, что некто не только компетентен в своей области, но и коммуникабелен и вообще полезен.

Вклад в проекты Open Source

Отличный способ обучения — продвижение проектов реализации с открытым исходным кодом: мало того что вы столкнетесь с трудностями, которые активизируют ваши способности, ваш код будут просматривать партнеры по сообществу, что сделает вас лучшим программистом. Это также будет прекрасно выглядеть в резюме.

Если вы новичок, то наилучшее для вас — это помощь кому-либо в составлении документации. Многие проекты с открытым исходным кодом страдают в части документации, а вы как новичок находитесь в превосходной позиции: вы можете что-то изучить, а затем объяснить это способом, который будет полезен для других новичков.

Иногда сообщество реализации с открытым исходным кодом может быть агрессивным, но если вы будете последовательны и открыты для конструктивной критики, то заметите, что ваш вклад приветствуется. Начните с чтения превосходной статьи *Bringing Kindness Back to Open Source* (`http://www.hanselman.com/blog/BringKindnessBackToOpenSource.aspx`) в блоге Скотта Хансельмана (Scot Hanselman). В ней автор рекомендует веб-сайт *Up for Grabs* (`http://up-for-grabs.net`), который помогает объединить программистов для проектов реализации с открытым исходным кодом. Выполните поиск по ключевому слову "JavaScript", и вы найдете множество проектов реализации с открытым исходным кодом, которым нужна помощь.

Заключение

Поздравляю вас с началом пути к профессии разработчика JavaScript! Часть материала в этой книге очень сложна, но надеюсь, вы уделили достаточно времени, чтобы в нем разобраться, и теперь ваши знания этого важного языка должны быть четко упорядочены. Если остались сложные моменты, не пугайтесь! JavaScript — сложный и мощный язык, который нельзя изучить быстро (даже в течение года). Если вы новичок в программировании, то в будущем вы, вероятно, захотите повторить часть материала этой книги; вы найдете новую суть в материале, который вначале казался вам сложным.

Спецификация ES6 породила новое поколение программистов, а вместе с ними и многие замечательные идеи. Я рекомендую вам читать все, что сможете, говорить с каждым программистом JavaScript, с которым сможете, и изучать каждый первоисточник, который сможете найти. В сообществе разработчиков JavaScript ожидается взрыв творческого потенциала, и я искренне надеюсь, что вы будете его частью.

Зарезервированные ключевые слова

Перечисленные ниже ключевые слова не могут использоваться в качестве идентификаторов (имен переменных, констант, свойств или функций) JavaScript.

- `await` (зарезервировано для использования в будущем)
- `break`
- `case`
- `class`
- `catch`
- `const`
- `continue`
- `debugger`
- `default`
- `delete`
- `do`
- `else`
- `enum` (зарезервировано для использования в будущем)
- `export`
- `extends`
- `false` (литеральное значение)
- `finally`
- `for`
- `function`

- `if`
- `implements` (зарезервировано для использования в будущем)
- `import`
- `in`
- `instanceof`
- `interface` (зарезервировано для использования в будущем)
- `let`
- `new`
- `null` (литеральное значение)
- `package` (зарезервировано для использования в будущем)
- `private` (зарезервировано для использования в будущем)
- `protectd` (зарезервировано для использования в будущем)
- `public` (зарезервировано для использования в будущем)
- `return`
- `super`
- `static` (зарезервировано для использования в будущем)
- `switch`
- `this`
- `throw`
- `true` (литеральное значение)
- `try`
- `typeof`
- `var`
- `void`
- `while`
- `with`
- `yield`

Перечисленные ниже ключевые слова были зарезервированы в спецификациях ECMAScript 1–3. Сейчас эти слова больше не зарезервированы, но я не рекомендую их использовать, поскольку в некоторых реализациях JavaScript они могут неправильно считаться зарезервированными.

- abstract
- boolean
- byte
- char
- double
- final
- float
- goto
- int
- long
- native
- short
- synchronized
- transient
- volatile

Приоритет операторов

Содержимое табл. Б.1 взято из документации *Mozilla Developer Network* и приведено для справки. Операторы спецификации ES7 опущены.

Таблица Б.1. Приоритет операторов от самого высокого (19) до самого низкого (0)

Приоритет	Тип оператора	Ассоциативность	Операторы
19	Группировка	Не определена	`(...)`
18	Доступ к члену	Справа налево	`... . ...`
	Вычисляемый доступ к члену	Справа налево	`...[...]`
	`new` (со списком аргументов)	Не определена	`new...(...)`
17	Вызов функции	Справа налево	`...(...)`
	`new` (без списка аргументов)	Справа налево	`new...`
16	Постфиксный инкремент	Не определена	`...++`
	Постфиксный декремент	Не определена	`...--`
15	Логическое NOT	Справа налево	`!...`
	Побитовое NOT	Справа налево	`~...`
	Унарная сумма	Справа налево	`+...`
	Унарное вычитание	Справа налево	`-...`
	Префиксный инкремент	Справа налево	`++...`
	Префиксный декремент	Справа налево	`--...`
	`typeof`	Справа налево	`typeof...`
	`void`	Справа налево	`void...`
	`delete`	Справа налево	`delete...`
14	Умножение	Справа налево	`...*...`
	Деление	Справа налево	`.../...`
	Остаток	Справа налево	`...%...`
13	Сложение	Справа налево	`...+...`
	Вычитание	Справа налево	`...-...`
12	Бинарный оператор сдвига влево	Справа налево	`...<<...`

Приоритет	Тип оператора	Ассоциативность	Операторы
	Бинарный оператор сдвига вправо	Справа налево	`...>>...`
	Бинарный оператор беззнакового сдвига вправо	Справа налево	`...>>>...`
11	Меньше	Справа налево	`...<...`
	Меньше или равно	Справа налево	`...<=...`
	Больше	Справа налево	`...>...`
	Больше или равно	Справа налево	`...>=...`
	`in`	Справа налево	`...in...`
	`instanceof`	Справа налево	`...instanceof...`
10	Равенство	Справа налево	`...==...`
	Неравенство	Справа налево	`...!=...`
	Строгое равенство	Справа налево	`...===...`
	Строгое неравенство	Справа налево	`...!==...`
9	Побитовое AND	Справа налево	`...&...`
8	Побитовое XOR	Справа налево	`...^...`
7	Побитовое OR	Справа налево	`...|...`
6	Логическое AND	Справа налево	`...&&...`
5	Логическое OR	Справа налево	`...||...`
4	Условное выражение	Справа налево	`...? ...:...`
3	Присваивание	Справа налево	`...=...`
			`...+=...`
			`...-=...`
			`...*=...`
			`.../=...`
			`...%=...`
			`...<<=...`
			`...>>=...`
			`...>>>=...`
			`...&=...`
			`...^=...`
			`...|=...`
2	`yield`	Справа налево	`yield...`
1	Расширение	Не определена	`... ...`
0	Запятая	Справа налево	`...,...`

Предметный указатель

Л

М

Н

О

П

СЕКРЕТЫ JAVASCRIPT НИНДЗЯ

ДЖОН РЕЗИГ
БЕЭР БИБО

Эта книга раскрывает секреты мастерства разработки веб-приложений на JavaScript. Начиная с пояснения таких основных понятий, как функции, объекты, замыкания, прототипы, регулярные выражения и таймеры, авторы постепенно проводят читателя по пути обучения от ученика до мастера, раскрывая немало секретов и специальных приемов программирования на конкретных примерах кода JavaScript. В книге уделяется немало внимания вопросам написания кросс-браузерного кода и преодолению связанных с этим типичных затруднений, что может принести немалую пользу всем, кто занимается разработкой веб-приложений. Книга рассчитана на подготовленных читателей, стремящихся повысить свой уровень мастерства в программировании на JavaScript в частности и разработке веб-приложений вообще.

www.williamspublishing.com

ISBN 978-5-8459-1959-5 **в продаже**