

«Вряд ли вы сможете найти более глубокий
или вдумчивый анализ ES6».

Ангус Кролл, автор книги
«If Hemingway Wrote JavaScript»

КАЙЛ СИМПСОН

ES6 & НЕ ТОЛЬКО

ВЫ НЕ ЗНАЕТЕ
JS



You Don't Know JS: ES6 and Beyond

Kyle Simpson

Beijing • Boston • Farnham • Sebastopol • Tokyo

O'REILLY®

O'REILLY®

КАЙЛ СИМПСОН

ES6 &

НЕ ТОЛЬКО

ВЫ НЕ ЗНАЕТЕ
JS



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

2017

ББК 32.988.02-018

УДК 004.738.5

С37

Симпсон К.

С37 ES6 и не только. — СПб.: Питер, 2017. — 336 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-496-02445-7

Даже если у вас уже есть опыт работы с JavaScript, скорее всего, язык вы в полной мере не знаете. Особое внимание в этой книге уделяется новым функциям, появившимся в EcmaScript 6 (ES6) — последней версии стандарта JavaScript.

ES6 повествует о тонкостях языка, малоизвестных большинству работающих на JavaScript программистов. Вооружившись этими знаниями, вы достигнете подлинного мастерства; выучите новый синтаксис; научитесь корректно использовать итераторы, генераторы, модули и классы; сможете более эффективно работать с данными; познакомитесь с новыми API, например Array, Object, Math, Number и String; расширите функционал программ с помощью мета-программирования.

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

ББК 32.988.02-018

УДК 004.738.5

Права на издание получены по соглашению с O'Reilly. Все права защищены. Никакая часть данной книги не может быть воспроизведена в какой бы то ни было форме без письменного разрешения владельцев авторских прав.

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1491904244 англ.

© 2016 Piter Press Ltd.

Authorized Russian translation of the English edition of You Don't Know JS: ES6 & Beyond, ISBN 9781491904244

© 2016 Getify Solutions, Inc

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

ISBN 978-5-496-02445-7

© Перевод на русский язык ООО Издательство «Питер», 2017

© Издание на русском языке, оформление ООО Издательство «Питер», 2017

© Серия «Бестселлеры O'Reilly», 2017

Оглавление

Введение	8
Предисловие	10
Цели и задачи.	12
Обзор.	13
Условные обозначения.	13
Использование примеров кода.	14
Safari® Books в Интернете.	15
От издательства.	16
Глава 1. ES: современность и будущее	17
Поддержка версий.	19
Транскомпиляция.	20
Подводим итоги.	23
Глава 2. Синтаксис	24
Объявления на уровне блоков кода.	25
Операторы Spread и Rest.	35
Значения параметров по умолчанию.	38
Деструктурирующее присваивание.	44
Расширения объектных литералов.	65
Шаблонные строки.	76
Стрелочные функции.	85
Цикл for..of.	94
Регулярные выражения.	97

Расширения числовых литералов	107
Unicode	109
Тип данных Symbol	117
Подводим итоги	124
Глава 3. Структура	126
Итераторы	126
Генераторы.	140
Модули	162
Классы	186
Подводим итоги	200
Глава 4. Управление асинхронными операциями	202
Обещания.	202
Генераторы и обещания	212
Подводим итоги	215
Глава 5. Коллекции	217
TypedArrays.	218
Карты	224
Объекты WeakMap	230
Объекты Set	231
WeakSets	234
Подводим итоги	234
Глава 6. Дополнения к API	236
Массив	236
Объект	249
Объект Math	254
Объект Number	256
Объект String	261
Подводим итоги	263

Глава 7. Метaproгpаммиpование 265

Имена функций	266
Метасвойства	269
Известные символы	270
Прокси	279
Reflect API.	296
Тестирование функциональных особенностей	301
Оптимизация хвостовой рекурсии	305
Подводим итоги	315

Глава 8. За пределами ES6 317

Асинхронные функции	318
Метод Object.observe(..)	323
Оператор возведения в степень	327
Свойства объектов и оператор	328
Метод Array#includes(..)	329
Принцип SIMD	330
Язык WebAssembly (WASM)	331
Подводим итоги	334

Введение

Кайл Симпсон — законченный прагматик.

На мой взгляд, нет высшей похвалы. Я считаю, что это самое важное качество, которым *должен* обладать разработчик программного обеспечения. Именно так: *должен*. Никто не умеет разделять слои языка JavaScript, превращая их в понятные и осмысленные фрагменты, так, как это делает Кайл.

Книга «ES6 и за его пределами» относится к уже известной читателям серии *You Don't Know JS*, а следовательно, вас ждет глубокое погружение в тему. Автор рассмотрит как очевидные, так и нетипичные случаи, раскрывающие семантику, которая или принимается как должно, или даже не рассматривается. То, о чем рассказывали предыдущие книги серии *You Don't Know JS*, в той или иной степени было знакомо читателям. Они или видели описываемые вещи, или слышали о них, или даже сталкивались на собственном опыте. Эта же книга содержит материал, практически не известный сообществу разработчиков, поскольку спецификация ECMAScript 2015 привела к революционным изменениям в языке JavaScript.

На протяжении последних лет я наблюдал, как Кайл стремится постичь новый материал, поднимаясь к вершинам, открытым лишь

небольшому числу его коллег. Задача, которую он поставил перед собой, была крайне непростой, ведь тогда еще не вышла в свет официальная спецификация языка. Я говорю чистую правду, и я прочитал от и до все, что Кайл написал для своей книги. Я отслеживал все изменения и наблюдал, как его код становился лучше, что свидетельствовало о растущем уровне понимания.

Эта книга коренным образом изменит ваше восприятие и откроет вам множество новых, доселе не известных вещей. Автор писал ее с намерением углубить ваши знания и одновременно расширить инструментарий. С ней вы уверенно вступите в новую эпоху программирования на языке JavaScript.

Рик Уолдрон (@rwaldron),
инженер открытых интернет-проектов в фирме Vocusp,
представитель Ecma/TC39,
для сайта jQuery

Предисловие

Уверен, что вы обратили внимание на «JS» в названии серии. Это вовсе не первые буквы тех слов, которые мы обычно произносим, когда хотим нелицеприятно высказаться по поводу JavaScript, — ведь все мы постоянно ругаемся на странности этого языка.

С момента появления Всемирной паутины JavaScript был основной технологией, обеспечивающей интерактивное взаимодействие с информацией. И если поначалу JavaScript ассоциировался с такими вещами, как тянущийся за указателем мыши мерцающий шлейф и надоедливые всплывающие окна, то за два десятилетия возможности языка возросли на много порядков, и сегодня вряд ли кто-то усомнится в его важности для функционирования Интернета в целом.

Тем не менее JavaScript всегда был мишенью для неумеренной критики, отчасти из-за своих странностей, но в основном из-за присущих языку особенностей проектирования. Даже само его имя, как однажды выразился Брендан Эйх, заставляет думать, что JavaScript — это непутевый младший брат более совершенного языка Java. Другое дело, что назвали его так практически случайно, по политическим и маркетинговым соображениям. Эти два языка

сильно отличаются во многих важных аспектах. «JavaScript» имеет такое же отношение к «Java», как карнавал к автомобилю¹.

Так как концепцию и синтаксические особенности JavaScript унаследовал от нескольких языков — например, бросающиеся в глаза программные конструкции в стиле С или менее очевидные принципы функционального программирования в стиле Scheme/Lisp, — он оказался понятен множеству людей, в том числе не имеющим серьезного опыта в разработке приложений. Демонстрирующая возможности языка программа «Hello World» на JavaScript пишется чрезвычайно просто, что делает его крайне привлекательным и легким для освоения.

Язык JavaScript прост, он позволяет быстро освоить теоретические основы и приступить к программированию, но из-за его странностей достичь в нем мастерства намного сложнее, чем в других языках. В ситуациях, когда для написания полноценной программы на С или С++ требуется глубокое понимание этих языков, JavaScript зачастую позволяет обойтись поверхностными знаниями.

Сложные концепции этого языка представлены *обманчиво* упрощенными способами, такими как передача функций в виде обратных вызовов, что провоцирует разработчиков применять инструменты JavaScript, не задумываясь о том, как именно они работают.

Этот язык популярен, легок в освоении и прост в использовании, но одновременно и сложен, так что без тщательного изучения его глубинных механизмов смысл происходящего будет ускользать даже от самых опытных JavaScript-разработчиков.

Вот в чем заключается парадокс JavaScript, вот где его ахиллесова пята. Изучением всех сложностей такого рода мы с вами и займемся — ведь если пользоваться JavaScript, не вникая в то, как он работает, можно так и не понять принцип его функционирования.

¹ Игра слов: carnival и car (англ.). — Примеч. пер.

Цели и задачи

Если вы склонны заносить в черный список все, что в JavaScript кажется странным или непонятным (а некоторые привыкли поступать именно так), в какой-то момент от богатого возможностями языка у вас останется лишь пустая оболочка.

Такое доступное всем подмножество механизмов JavaScript принято считать сильными сторонами этого языка, но правильнее назвать это легкими в освоении, безопасными или даже минимальными возможностями.

Я предлагаю вам поступить наоборот: досконально изучить JavaScript, чтобы понять даже самые сложные его особенности. Именно о них пойдет речь в этой книге.

Мне известна склонность JS-разработчиков изучать лишь минимум, необходимый для решения конкретной задачи, но в моей книге вы не встретите распространенной рекомендации избегать сложностей.

Даже если что-то работает нужным мне образом, я не готов удовлетвориться самим этим фактом — мне важно понять, почему и как оно работает. Хотелось бы, чтобы вы разделили мой подход. Я ненавязчиво зову вас пройти по тернистой дороге, которой мало кто ходил, и полностью осмыслить, что представляет собой язык JavaScript и какие возможности он дает. И когда вы будете обладать этими знаниями, ни одна техника, ни одна платформа, ни один новый подход не окажутся за пределами вашего понимания.

Каждая из книг серии *You Don't Know JS* глубоко и исчерпывающе раскрывает конкретные ключевые элементы языка, которые зачастую толкуются неверно или поверхностно. После прочтения этой литературы вы получите твердую уверенность в том, что понимаете не только теоретические, но и практические нюансы.

Те познания в JavaScript, что у вас есть, скорее всего, вы получили от людей, которые сами попали в ловушку недостаточного понимания. Это всего лишь тень того, чем JavaScript является на самом

деле. Вы еще толком не знаете его, но книги серии *You Don't Know JS* помогут вам наверстать упущенное. Поэтому вперед, дорогие друзья, язык JavaScript ждет вас!

Обзор

Язык JavaScript — обширная тема. Его легко изучить поверхностно и намного сложнее освоить полностью (или хотя бы в скольконибудь серьезном объеме). Сталкиваясь с чем-то непонятным, разработчики, как правило, списывают это на несовершенство языка, не допуская мысли о собственной неосведомленности. Книги нашей серии способны исправить ситуацию: вы в полной мере оцените возможности языка, когда начнете по-настоящему разбираться в нем.



Многие приведенные в тексте примеры рассчитаны на современные (и будущие) реализации JavaScript, такие как ES6. Некоторые фрагменты кода могут работать по-разному при запуске в новых и старых (использовавшихся до ES6) версиях интерпретаторов.

Условные обозначения

В книге используются следующие условные обозначения:

Курсив

Указывает на новые термины, URL, адреса электронной почты, имена и расширения файлов.

Моноширинный шрифт

Используется в листингах, а также в обычном тексте для обозначения элементов программного кода, таких как имена переменных и функций, баз данных, типов данных, переменных среды, операторов и ключевых слов.

Моноширинный жирный шрифт

Указывает на команды или другой текст, который должен быть набран пользователем.

Моноширинный курсив

Указывает на фрагменты текста, которые нужно заменить пользовательскими или определяемыми контекстом значениями.



Этот элемент обозначает рекомендацию.



Этот элемент обозначает общее примечание.



Этот элемент обозначает предупреждение.

Использование примеров кода

Дополнительные материалы (примеры кода, упражнения и пр.) доступны для скачивания по адресу <http://bit.ly/ydkjs-es6beyond-code>.

Эта книга призвана помочь вам в работе. В общем случае все приведенные в ней примеры вы можете использовать в своих разработках и документации. Связываться с нами для получения разрешения необходимо только в случаях, когда вы хотите скопировать себе значительные фрагменты текста. Так, если вы решите вставить несколько строк кода из книги в свою программу, спрашивать на это разрешения вам не придется. А вот если вы будете продавать или другим способом распространять диски с материалами, взятыми из книг издательства O'Reilly, тогда вам сперва придется связаться с нами. Отвечать на вопросы, цитируя текст и программный

код из книги, вполне допустимо, а вот вставлять те же материалы в значительных объемах в документацию к вашей продукции можно только с разрешения издательства.

Мы не требуем, чтобы вы всегда ссылались на источник материалов, но будем крайне признательны, если вы это сделаете. Ссылка обычно включает в себя название книги, указание автора и издателя, а также ISBN. В данном случае она выглядит так: «You Don't Know JavaScript: ES6 & Beyond by Kyle Simpson (O'Reilly). Copyright 2016 Getify Solutions, Inc., 978-1-491-90424-4».

Если вас интересует использование примеров, выходящее за рамки данных выше разрешений, обращайтесь по адресу permissions@oreilly.com.

Safari® Books в Интернете



Safari Books Online (<https://www.safaribooksonline.com/>) — это цифровая библиотека, предоставляющая в формате книг и видеозаписей материалы (<https://www.safaribooksonline.com/explore/>) ведущих мировых экспертов в области технологий и бизнеса.

Технические специалисты, разработчики программного обеспечения, веб-дизайнеры, предприниматели и люди творческих профессий используют сайт Safari Books Online как основной ресурс для исследований, решения задач, обучения и проведения сертификационных тренингов.

Сайт предлагает различные варианты подписки для компаний (<https://www.safaribooksonline.com/enterprise/>), государственных учреждений (<https://www.safaribooksonline.com/government/>), учебных заведений (<https://www.safaribooksonline.com/academic-public-library/>) и физических лиц.

Покупка членства дает доступ к тысячам книг, обучающих видеороликов и еще не опубликованных материалов, собранных в единую

базу данных с полнофункциональным поиском. Там есть книги таких издательств, как O'Reilly Media, Prentice Hall Professional, Addison Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology и сотен других (<https://www.safaribooksonline.com/our-library/>). Дополнительную информацию вы найдете на сайте Safari Books Online.

От издательства

На веб-странице этой книги по адресу <http://bit.ly/ydkjs-es6-beyond> вы найдете сведения об опечатках, список примеров и дополнительные материалы.

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

Мы будем рады узнать ваше мнение!

Все исходные тексты, приведенные в книге, вы можете найти по адресу <http://www.piter.com>.

На веб-сайте издательства <http://www.piter.com> вы найдете подробную информацию о наших книгах.

1 ES: современность и будущее

Для чтения этой книги вы должны хорошо владеть языком JavaScript вплоть до последнего (на момент написания книги) стандарта, который называется ES5 (точнее, ES5.1), поскольку мы с вами будем рассматривать новый стандарт ES6, попутно пытаясь понять, какие перспективы ждут JS.

Если вы не очень уверены в своих знаниях JavaScript, рекомендую предварительно ознакомиться с предшествующими книгами серии *You Don't Know JS*.

- *Up & Going*: Вы только начинаете изучать программирование и JS? Перед вами карта, которая поможет вам в путешествии по новой области знаний.
- *Scope & Closures*: Известно ли вам, что в основе лексического контекста JS лежит семантика компилятора (а не интерпретатора)? Можете ли вы объяснить, каким образом замыкания являются прямым результатом лексической области видимости и функций как значений?
- *this & Object Prototypes*: Можете ли вы назвать четыре варианта значения ключевого слова `this` в зависимости от контекста вызова? Приходилось ли вам путаться в псевдоклас-

сах JS, вместо того чтобы воспользоваться более простым шаблоном проектирования *behavior delegation*? А слышали ли вы когда-нибудь про объекты, связанные с другими объектами (OLOO)?

- *Types & Grammar*: Знакомы ли вы со встроенными типами в JS и, что более важно, знаете ли способы корректного и безопасного приведения типов? Насколько уверенно вы разбираетесь в нюансах грамматики и синтаксиса этого языка?
- *Async & Performance*: Вы все еще используете обратные вызовы для управления асинхронными действиями? А можете ли вы объяснить, что такое объект *promise* и как он позволяет избежать ситуации, когда каждая фоновая операция возвращает свой результат (или ошибку) в обратном вызове? Знаете ли вы, как с помощью генераторов улучшить читабельность асинхронного кода? Наконец, известно ли вам, что представляет собой полноценная оптимизация JS-программ и отдельных операций?

Если вы уже прочитали все эти книги и освоили рассматриваемые там темы, значит, пришло время погрузиться в эволюцию языка JS и исследовать перемены, которые ждут нас как в ближайшее время, так и в отдаленной перспективе.

В отличие от предыдущего стандарта, ES6 нельзя назвать еще одним скромным набором добавленных к языку API. Он принес с собой множество новых синтаксических форм, и к некоторым из них, вполне возможно, будет не так-то просто привыкнуть. Появились также новые структуры и новые вспомогательные модули API для различных типов данных.

ES6 — это шаг далеко вперед. Даже если вы считаете, что хорошо знаете JS стандарта ES5, вы столкнетесь с множеством незнакомых вещей, так что будьте готовы! В книге рассмотрены все основные нововведения ES6, без которых невозможно войти в курс дела, а также дан краткий обзор планируемых функций — о них имеет смысл знать уже сейчас.



Весь приведенный в книге код рассчитан на среду исполнения ES6+. На момент написания этих строк уровень поддержки ES6 в браузерах и в JS-средах (таких, как Node.js) несколько разнился, так что вы можете обнаружить, что полученный вами результат отличается от описанного.

Поддержка версий

Стандарт JavaScript официально называется ECMAScript (или сокращенно ES), и до недавнего времени все его версии обозначались только целыми числами. ES1 и ES2 не получили известности и массовой реализации. Первой широко распространившейся основой для JavaScript стал ES3 — стандарт этого языка для браузеров Internet Explorer с 6-й по 8-ю версию и для мобильных браузеров Android 2.x. По политическим причинам, о которых я умолчу, злополучная версия ES4 так и не увидела света.

В 2009 году был официально завершен ES5 (ES5.1 появился в 2011-м), получивший распространение в качестве стандарта для множества современных браузеров, таких как Firefox, Chrome, Opera, Safari и др.

Следующая версия JS (появление которой было перенесено с 2013-го сначала на 2014-й, а затем на 2015 год) в обсуждениях фигурировала под очевидным именем ES6. Но позднее стали поступать предложения перейти к схеме именования, основанной на годе выхода очередной версии, например ES2016 (она же ES7), которая будет закончена до конца 2016 года. Согласны с таким подходом далеко не все, но есть вероятность, что стандарт ES6 станет известен пользователям под названием ES2015. А появление версии ES2016 станет свидетельством окончательного перехода на новую схему именования.

Кроме того, было отмечено, что скорость эволюции JS превышает одну версию в год. Как только в обсуждениях стандарта возникает новая идея, разработчики браузеров предлагают прототипы нового

функционала, а программисты-первопроходцы принимаются экспериментировать с кодом.

Обычно задолго до официального одобрения новый функционал становится стандартом де-факто благодаря ранним прототипам движка и инструментария. Соответственно, имеет смысл рассматривать будущие версии JS как связанные с появлением нового функционала, а не с произвольным набором основных особенностей (как делается сейчас) или с годом (как планируется).

В этом случае номер версии перестает иметь ту важность, которой обладал раньше, а JavaScript превращается в живой, постоянно меняющийся стандарт. И лучше не говорить о коде как о «написанном в соответствии с таким-то стандартом», а рассматривать его в зависимости от поддерживаемых функциональных особенностей.

Транскомпиляция

Быстрая эволюция функционала ставит серьезную проблему перед разработчиками, желающими использовать новые возможности в ситуации, когда их сайты или приложения работают в более старых браузерах, не поддерживающих нововведения.

По всей видимости, ES5 не прижился во многих местах, потому что в основном базу кода не приводили в соответствие с новым стандартом до тех пор, пока не прекратилась поддержка большинства, если не всех, предшествующих платформ. В результате многие разработчики только недавно начали пользоваться такими вещами, как, к примеру, строгий режим, появившийся в ES5 более пяти лет назад.

Подобные многолетние промедления повсеместно считаются вредными для будущего экосистемы JS. Люди, занимающиеся развитием языка, мечтают, чтобы разработчики начинали создавать код с учетом новых функциональных особенностей и шаблонов, сразу

же после того, как будет утверждена спецификация, и браузеры смогут все это реализовывать.

Как же разрешить противоречие? Здесь на помощь приходят специальные инструменты, в частности техника *транскомпиляции*¹. Грубо говоря, вы посредством специального инструмента преобразуете код ES6 в эквивалент (или нечто близкое к таковому), работающий в окружениях ES5.

В качестве примера возьмем сокращенные определения свойства (см. раздел «Расширения объектных литералов» в главе 2). Вот как это делается в ES6:

```
var    foo = [1,2,3];
var    obj = {
  foo   // означает 'foo: foo'
};

obj.foo;    // [1,2,3]
```

А вот каким образом (примерно) он транскомпилируется:

```
var    foo = [1,2,3];

var    obj = {
  foo: foo
};

obj.foo;    // [1,2,3]
```

Такое небольшое, но удобное преобразование позволяет в случае одинаковых имен сократить объявление объектного литерала `foo: foo` до `foo`. Действия транскомпилятора в этом случае представляют собой встроенный рабочий процесс, аналогичный линтингу, минификации и другим подобным операциям.

¹ Transpiling — термин образован от transformation (преобразование) и compiling (компиляция) (англ.). — *Примеч. пер.*

Библиотеки Shim (полизаполнения)

Далеко не всем новым функциональным особенностям ES6 требуется транскompилятор. *Полизаполнения* (polyfills), которые также называют библиотеками Shim, представляют собой шаблоны для определения поведений из новой среды для более старых сред. В синтаксисе полизаполнения недопустимы, но для различных API их вполне можно использовать.

Давайте рассмотрим новый метод `Object.is(..)`, предназначенный для проверки строгой эквивалентности двух значений, но без подробных исключений, которые есть у оператора `===` для значений `NaN` и `-0`. Полизаполнение для метода `Object.is(..)` создается очень просто:

```
if (!Object.is) {  
  Object.is = function(v1, v2) {  
    // проверка для значения '-0'  
    if (v1 === 0 && v2 === 0) {  
      return 1 / v1 === 1 / v2;  
    }  
    // проверка для значения 'NaN'  
    if (v1 !== v1) {  
      return v2 !== v2;  
    }  
    // все остальное  
    return v1 === v2;  
  };  
}
```



Обратите внимание на внешнее граничное условие оператора `if`, охватывающее полизаполнение. Это важная деталь, означающая, что резервное поведение данный фрагмент кода включает только в более старых контекстах, где рассматриваемый API еще не определен; необходимости переписывать существующий API практически никогда не возникает.

Есть замечательная коллекция ES6 Shim (<https://github.com/paulmillr/es6-shim/>), которую стоит включать во все новые JS-проекты.

Предполагается, что JS ждет непрерывное развитие и что поддержка в браузерах новых функций будет реализовываться постепенно по мере их появления, а не большими фрагментами. Так что самая лучшая стратегия сохранения актуальности — это добавление в базу кода полизаполнений, включение транскомпиляции в процесс сборки и постоянная готовность самого разработчика к изменениям.

Те же, кто мыслит консервативно и откладывает использование нового функционала, пока не исчезнут все работающие без него браузеры, всегда будут плестись далеко позади. Их обойдут стороной все инновации, позволяющие сделать написание кода на JavaScript более результативным, рациональным и надежным.

Подводим итоги

На момент написания книги стандарт ES6 (кто-то называет его ES2015) только появился, поэтому вам предстоит многому научиться.

Однако куда важнее перестроить свое мировоззрение в соответствии с новым вариантом развития языка JavaScript. Обыкновенно годами ждать официальных документов, одобряющих смену стандарта, должно остаться в прошлом.

Теперь новые функциональные особенности JavaScript сразу же после своего появления реализуются в браузерах, и только от вас зависит, начнете вы пользоваться ими немедленно или же продолжите действовать неэффективно в попытках запрыгнуть в уходящий поезд.

Неважно, какие еще формы примет JavaScript, — теперь это будет происходить быстрее, чем когда-либо в прошлом. Транскомпиляторы и полизаполнения — вот инструменты, которые позволят вам все время оставаться на переднем крае развития языка.

Вы должны принять новую реальность JavaScript, где разработчикам настоятельно рекомендуется перейти от выжидания к активной позиции. А начнется все с изучения ES6.

2

Синтаксис

Если у вас есть хоть какой-то опыт написания программ на JS, скорее всего, вы достаточно хорошо знакомы с его синтаксисом. У него немало своих особенностей, но несмотря на это он достаточно рационален и несложен, а кроме того, имеет множество аналогий в других языках.

Стандарт ES6 добавляет ряд новых синтаксических форм, к которым вам нужно будет привыкнуть. О них я расскажу в этой главе, чтобы дать представление о том, с чем вам предстоит работать.



На момент написания этой книги некоторые составляющие нового функционала уже были полноценно реализованы в браузерах (Firefox, Chrome и др.), другие — реализованы лишь частично, а какие-то — вообще пока недоступны нигде. Соответственно, если вы будете использовать приведенные в книге примеры как есть, результаты могут оказаться непредсказуемыми, так что лучше вам прибегнуть к транскompиляторам, умеющим работать с большей частью новых функциональных особенностей.

Например, существует замечательная, легкая в использовании «песочница» ES6Fiddle (<http://www.es6fiddle.net/>) для запуска ES6-кода прямо в браузере, а также онлайн-овая REPL-среда для транскompилятора Babel (<http://babeljs.io/repl/>).

Объявления на уровне блоков кода

Вы, наверное, знаете, что областью видимости переменной в JavaScript в основном всегда была функция. Предпочтительным способом задания видимости переменной в определенном блоке кода, кроме обычного объявления функции, является немедленно вызываемая функция (IIFE). Например:

```
var a = 2;

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

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

Оператор let

Впрочем, теперь у нас есть возможность создать объявление, связанное с произвольным блоком, которое вполне логично называется *блочной областью видимости* (block scoping). Для этого достаточно будет пары фигурных скобок { .. }. Вместо оператора var, объявляющего область видимости переменных внутри функции, в которую он вложен (или глобальную область видимости, если он находится на верхнем уровне), мы воспользуемся оператором let:

```
var a = 2;

{
    let a = 3;
    console.log( a );    // 3
}

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

Использование отдельных блоков { .. } — не очень распространенная практика в JS, но это всегда работает. Если вы имели дело

с языками, в которых возможна область видимости на уровне блоков, вы легко распознаете данный шаблон.

На мой взгляд, это самый лучший способ создания переменных с блочной областью видимости. Оператор `let` всегда должен находиться в верхней части блока, причем желательно, чтобы он был один, вне зависимости от количества объявляемых переменных.

Если говорить о стилистике, то я считаю, что правильнее помещать оператор `let` на одну строчку с открывающейся фигурной скобкой `{`, как бы подчеркивая, что единственное назначение блока — это определение области видимости переменных.

```
{ let a = 2, b, c;  
  // ..  
}
```

Я понимаю, что написанное мной выглядит странно и, скорее всего, противоречит рекомендациям, которые даются в других книгах по ES6. Но у меня есть на это веские причины.

Существует еще одна экспериментальная (не входящая в стандарт) форма объявления при помощи оператора `let`, называемая `let-блоком`. Она выглядит следующим образом:

```
let (a = 2, b, c) {  
  // ..  
}
```

Я называю эту форму *явным* объявлением области видимости внутри блока. Имитирующая оператор `var` форма объявления переменной `let ..` более *неявна* — она как бы захватывает все содержимое фигурных скобок `{ .. }`. Как правило, разработчики считают *явные* механизмы более предпочтительными, и мне кажется, что в данном случае имеет смысл придерживаться этого подхода.

Если сравнивать два предыдущих фрагмента кода, в глаза бросается их сходство, и, с моей точки зрения, оба они стилистически

могут рассматриваться как *явное* объявление блочной области видимости. К сожалению, наиболее *явная* форма `let (..) { .. }` в стандарт ES6 не вошла. Существует вероятность, что она появится в последующих версиях ES6, пока же предыдущий пример — это лучшее, чем вы можете воспользоваться.

Подчеркнуть *неявную* природу объявления переменных с помощью оператора `let ..` можно так, как показано ниже:

```
let a = 2;

if (a > 1) {
  let b = a * 3;
  console.log( b );           // 6

  for (let i = a; i <= b; i++) {
    let j = i + 10;
    console.log( j );
  }
  // 12 13 14 15 16

  let c = a + b;
  console.log( c );           // 8
}
```

Попробуйте, не глядя на предыдущие фрагменты кода, ответить, какие переменные существуют только внутри оператора `if`, а какие — только внутри цикла `for`.

Правильный ответ: у `if` это переменные `b` и `c`, а у `for` — `i` и `j`.

Долго ли вы размышляли? И не показался ли вам странным тот факт, что переменная `i` не попала в область видимости оператора `if`? Причина заминки с ответом и возникших вопросов (или, как я обычно говорю, «ментальной нагрузки») появилась потому, что механизм работы оператора `let`, во-первых, непривычный для вас, а во-вторых, *неявный*.

Кстати, помещать объявление `let c = ..` так низко — небезопасно. Если переменные, объявленные с помощью оператора `var`, связываются со всей областью видимости функции вне зависимости

от места их расположения, то использование `let` связывает переменную с областью видимости блока, но инициализация произойдет только после объявления этой переменной. Попытки обращения к ней до инициализации вызовут ошибку, в то время как в случае оператора `var` порядок действий значения не имеет (если не брать во внимание стилистический аспект).

Смотрите:

```
{  
  console.log( a );      // значение не определено  
  console.log( b );      // ReferenceError!  
  
  var a;  
  let b;  
}
```



Ошибка `ReferenceError`, генерируемая при попытке слишком рано обратиться к переменной, которая ниже объявляется при помощи оператора `let`, технически называется ошибкой временной мертвой зоны (TDZ — temporal dead zone). Это значит, что вы хотите получить доступ к переменной, которая уже объявлена, но еще не инициализирована. Такая ситуация — не единственная, где появляется ошибка TDZ. Работая с JS стандарта ES6, вы еще не раз с ней столкнетесь. Кроме того, имейте в виду, что «инициализация» возможна и без явного присвоения значения, в частности достаточно выражения `let b;`. Переменная, которой во время объявления ничего не присваивается, по умолчанию получает значение `undefined`. Соответственно, выражение `let b;` эквивалентно выражению `let b = undefined;`. При этом вне зависимости от того, явным или неявным образом было присвоено значение, доступ к переменной `b` появится только после выполнения оператора `let b`.

Есть и еще один подводный камень: с переменными, вызывающими ошибку мертвой зоны, оператор `typeof` ведет себя не так, как с необъявленными (или объявленными) переменными. Например:

```
{  
  // переменная 'a' не объявлена  
  if (typeof a === "undefined") {  
    console.log( "cool" );  
  }  
  // переменная 'b' объявлена, но находится в мертвой зоне  
  if (typeof b === "undefined") {           // ReferenceError!  
    // ..  
  }  
  
  // ..  
  
  let b;  
}
```

Переменная **a** не была объявлена, так что единственный безопасный способ проверить, существует она или нет, — использовать оператор **typeof**. При этом запись **typeof b** приводит к появлению ошибки TDZ, так как строка **let b**, объявляющая переменную, располагается ниже.

Думаю, теперь вы понимаете, почему я настоятельно рекомендую выполнять все объявления с помощью оператора **let** в самой верхней строке блока. Только так можно избежать преждевременного обращения к переменным. Кроме того, при таком подходе сразу видно, какие переменные содержит тот или иной блок.

При этом поведение самого блока (операторов **if**, циклов **while** и т. п.) вовсе не обязательно должно совпадать с поведением области видимости.

Такая ясность, для достижения которой достаточно следовать кое-каким принципам, избавит вас от головной боли на стадии рефакторинга и от неожиданных проблем в будущем.



Более подробно оператор **let** и блочная область видимости рассматриваются в главе 3 книги *Scope & Closures*.

Оператор `let` и цикл `for`

Единственный случай, когда я советую отказаться от *явной* формы объявления блочной области видимости с помощью оператора `let`, — это его вставка в заголовок цикла `for`. Данный нюанс кому-то может показаться незначительным, но я считаю, что здесь имеет место важная функциональная особенность ES6.

Рассмотрим пример:

```
var funcs = [];  
  
for (let i = 0; i < 5; i++) {  
  funcs.push( function(){  
    console.log( i );  
  } );  
}  
  
funcs[3]();           // 3
```

Оператор `let i` в заголовке цикла `for` объявляет переменную `i` на каждой его итерации. Это означает, что создаваемые внутри цикла на каждой итерации замыкания переменные ведут себя именно так, как ожидается.

Если в том же самом фрагменте вставить в заголовок цикла `for` выражение `var i`, вы получите значение 5 вместо 3, так как в этом случае замыканию будет подвергаться только переменная `i` во внешней области видимости, а не новая `i` для каждой итерации.

Аналогичный результат дает более развернутый вариант кода:

```
var funcs = [];  
  
for (var i = 0; i < 5; i++) {  
  let j = i;  
  funcs.push( function(){  
    console.log( j );  
  } );  
}  
  
funcs[3]();           // 3
```

Здесь мы принудительно создаем на каждой итерации новую переменную `j`, а затем замыкание действует, как и в предыдущем случае. Мне больше нравится предыдущий подход с формой `for (let ..) ..`, дающий дополнительные возможности. Кто-то, вероятно, скажет, что он в некотором смысле неявен, но, с моей точки зрения, он достаточно явный и весьма полезный.

Аналогичным образом оператор `let` работает с циклами `for..in` и `for..of` (см. раздел «Цикл `for..of`» этой главы).

Объявления с оператором `const`

Существует еще одна возможность объявления с блочной областью видимости. Ее нам дает создающий *константы* оператор `const`.

Что такое константа? Это переменная, которая после присвоения начального значения доступна только для чтения. Например:

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

  a = 3;               // TypeError!
}
```

Вы не можете менять значение такой переменной после того, как задали его во время объявления. Объявление при помощи оператора `const` всегда должно сопровождаться инициализацией. Если вам требуется *константа* со значением `undefined`, напишите `const a = undefined`.

В случае констант ограничение накладывается не на их значение, а на операцию присваивания. Другими словами, значение оказывается неизменяемым не потому, что речь идет о константе, а из-за невозможности присваивания другого. Если значение комплексное, например, в случае массива или объекта, оно допускает модификацию:

```
{  
  const a = [1,2,3];  
  a.push( 4 );  
  console.log( a );    // [1,2,3,4]  
  
  a = 42;              // TypeError!  
}
```

Переменная `a` содержит не константный массив, а неизменяемую ссылку на него. Однако сам массив допускает модификацию.



Если вы присваиваете объект или массив как константу, это значение станет доступным сборщику мусора только после выхода из лексической области видимости константы, ведь отменить ссылку на такое значение нельзя. Иногда подобное поведение оказывается как раз тем, что нужно.

По сути, объявление констант открывает возможность, которая в течение многих лет реализовывалась стилистически, когда переменным с именами, состоящими исключительно из строчных букв, присваивался литерал. В случае оператора `var` присваивание значений не обязательно, в то время как `const` работает только в связке с присваиванием, что позволяет избежать случайных изменений.

Оператор `const` можно использовать для объявления переменных в циклах `for`, `for..in` и `for..of` (см. раздел «Цикл `for..of`»), но при любой попытке присвоить новое значение, например, типичным для циклов способом `i++`, будет появляться сообщение об ошибке.

Применение оператора `const`

Есть мнение, что в определенных сценариях оператор `const` лучше оптимизируется движком JS, чем операторы `let` или `var`: якобы движок, обнаружив, что значение/тип переменной меняться не будет, перестанет отслеживать часть аспектов.

Вне зависимости от истинности данного предположения перед программистом стоит важный выбор. Нужно решить, требуется ли вообще неизменяемое поведение или нет. Напоминаю, что исходный код следует писать таким образом, чтобы ваши намерения были четко понятны не только вам в момент работы над кодом, но и вам в будущем, а также всем вашим коллегам.

Некоторые разработчики предпочитают изначально объявлять все переменные как константы, а затем, если возникнет необходимость изменения, делать их модифицируемыми при помощи оператора `let`. Это интересный подход, но я не думаю, что он улучшает читабельность или осмысленность кода.

Многие рассматривают подобную практику как *защиту*, но дело обстоит не совсем так, ведь в будущем любой разработчик, решивший присвоить константе новое значение, может легко заменить в объявлении `const` на `let`. Так что в лучшем случае такой подход способен защитить от случайных изменений. Я еще раз напомню, что если отбросить в сторону нашу интуицию и чувства, объективных и четких критериев, определяющих случайные действия и способы их предотвращения, не существует. Аналогичные вопросы возникают вокруг принудительного присвоения типов.

Чтобы не делать свой код запутанным, я советую превращать в константы только те переменные, которые вы целенаправленно хотите сделать неизменяемыми. Другими словами, константами нужно пользоваться как инструментом, ясно сообщаящим о ваших намерениях.

Функции с областью видимости на уровне блока

Начиная с ES6 область видимости объявленных внутри блока функций ограничивается этим блоком. Ранее подобного поведения спецификация не предусматривала, хотя оно в любом случае появлялось во многих реализациях, так что произошло объединение спецификации и фактического положения дел.

Рассмотрим пример:

```
{
    foo();           // работаем!

    function foo() {
        // ..
    }
}

foo();              // ReferenceError
```

Функция `foo()` объявлена внутри блока `{ .. }`, и в соответствии со стандартом ES6 ее область видимости ограничивается этим блоком. Так что за его пределами она недоступна. Кстати, обратите внимание, что она поднята в верхнюю часть блока, что недопустимо при объявлении с помощью оператора `let`. Но в данном случае ошибки TDZ не возникает.

Блочная область видимости функции может стать проблемой, если раньше вы писали код вроде того, что приведен ниже, и до сих пор полагаетесь на устаревшее поведение.

```
if (something) {
    function foo() {
        console.log( "1" );
    }
}
else {
    function foo() {
        console.log( "2" );
    }
}

foo();           // ??
```

В средах, работающих с более ранними версиями языка, функция `foo()` даст значение «2» вне зависимости от результатов проверки условия `something`, так как оба объявления функции внутри

блока подняты вверх, и значение будет иметь только второй вариант.

В ES6 последняя строчка приведет к ошибке `ReferenceError`.

Операторы Spread и Rest

В ES6 появился новый оператор `...`, который в зависимости от контекста применения может интерпретироваться как оператор `spread` или как оператор `rest`. Рассмотрим пример:

```
function foo(x,y,z) {  
    console.log( x, y, z );  
}  
foo( ...[1,2,3] );    // 1 2 3
```

Когда оператор `...` стоит перед массивом (или любым другим набором доступных для перебора значений, которые будут рассматриваться в главе 3), он разделяет массив на отдельные значения.

Как правило, вариант, показанный в предыдущем фрагменте кода, используется, когда требуется превратить массив в набор аргументов для вызова функции. В этом случае оператор `...` дает более простую синтаксическую замену метода `apply(...)`, который использовался с той же целью до появления ES6.

```
foo.apply( null, [1,2,3] ); // 1 2 3
```

Впрочем, оператор `...` позволяет расширять значение и в других контекстах, например внутри объявления другого массива:

```
var    a = [2,3,4];  
var    b = [ 1, ...a, 5 ];  
  
console.log( b );    // [1,2,3,4,5]
```

В этом примере оператор `...`, по сути, заменяет метод `concat(...)`, потому что мы получаем такой же результат, как и в случае `[1].concat(a, [5])`.

Кроме того, оператор `...` повсеместно используется для противоположной цели: вместо разделения массива он *собирает* набор значений в массив. Например:

```
function foo(x, y, ...z) {  
    console.log( x, y, z );  
}  
  
foo( 1, 2, 3, 4, 5 );    // 1 2 [3,4,5]
```

Запись `...z` в этом фрагменте кода фактически представляет собой инструкцию «собрать остальные аргументы (если они есть) в массив с именем `z`». Так как переменной `x` присвоено значение 1, а переменной `y` — значение 2, остальные аргументы 3, 4 и 5 помещаются в массив `z`.

При отсутствии именованных параметров оператором `...` будут собраны все аргументы:

```
function foo(...args) {  
    console.log( args );  
}  
  
foo( 1, 2, 3, 4, 5 );    // [1,2,3,4,5]
```



Запись `...args` в объявлении функции `foo(...)` обычно называют «оставшимися параметрами», так как с ее помощью вы собираете в массив все оставшиеся элементы.

Но самое примечательное свойство оператора `...` — это возможность его применения в качестве альтернативы давным-давно устаревшему массиву `arguments` (который на самом деле псевдомассив). Так как объект `args` (или, как многие предпочитают его

называть, `r` или `rest`) представляет собой настоящий массив, можно уже не прибегать к многочисленным обходным приемам, которыми мы пользовались до появления ES6, чтобы превратить аргументы в нечто, позволяющее обращаться с собой как с массивом.

Рассмотрим пример:

```
// новый способ в соответствии со стандартом ES6
function foo(...args) {
    // 'args' это настоящий массив

    // отбрасываем первый элемент в массиве 'args'
    args.shift();

    // передаем все содержимое 'args' как аргументы
    // методу 'console.log(..)'
    console.log( ...args );
}

// старый, использовавшийся до появления ES6 способ
function bar() {
    // превращаем 'arguments' в настоящий массив
    var args = Array.prototype.slice.call( arguments );

    // добавляем в конец какие-то элементы
    args.push( 4, 5 );

    // убираем нечетные числа
    args = args.filter( function(v){
        return v % 2 == 0;
    } );

    // передаем все содержимое 'args' как аргументы
    // в функцию 'foo(..)'
    foo.apply( null, args );
}

bar( 0, 1, 2, 3 );    // 2 4
```

Оператор `...args` в объявлении функции `foo(..)` собирает аргументы, а в вызове метода `console.log(..)` разделяет их. Это хорошая

иллюстрация симметричного, но диаметрально противоположного по смыслу его применения.

Кроме того, вставка оператора `...` в объявление функции — еще один случай сбора значений, который мы рассмотрим ниже в разделе «Слишком много, слишком мало, в самый раз».

Значения параметров по умолчанию

Одна из самых распространенных идиом языка JavaScript связана с заданием значения по умолчанию для параметра функции. В течение многих лет мы делали это вот таким способом:

```
function foo(x,y) {  
    x = x || 11;  
    y = y || 31;  
  
    console.log( x + y );  
}  
  
foo();           // 42  
foo( 5, 6 );     // 11  
foo( 5 );        // 36  
foo( null, 6 );  // 17
```

Тот, кто пользовался этим шаблоном, разумеется, знает, что при всей своей полезности он до известной степени опасен. Скажем, возможна ситуация, когда нужно передать одному из параметров значение, которое интерпретируется как недействительное. Например:

```
foo( 0, 42 ); // 53 <-- Ой, не 42
```

Почему так произошло? Потому что логический эквивалент значения `0` это `false`, и, соответственно, выражение `x || 11`, дающее нам `11`, не может быть напрямую передано в `0`.

Для исправления ситуации некоторые ставят более развернутое условие:

```
function foo(x,y) {  
    x = (x !== undefined) ? x : 11;  
    y = (y !== undefined) ? y : 31;  
  
    console.log( x + y );  
}  
  
foo( 0, 42 );           // 42  
foo( undefined, 6 );    // 17
```

То есть можно непосредственно передать любое значение, кроме `undefined`, причем значение `undefined` сигнализирует, что параметр не был передан в функцию. Такой подход замечательно работает, пока не возникает необходимость передать в функцию именно это значение.

В таком случае можно проверить, действительно ли пропущен нужный аргумент. То есть мы должны удостовериться, что он физически отсутствует в массиве аргументов, например, вот таким образом:

```
function foo(x,y) {  
    x = (0 in arguments) ? x : 11;  
    y = (1 in arguments) ? y : 31;  
  
    console.log( x + y );  
}  
  
foo( 5 );               // 36  
foo( 5, undefined );    // NaN
```

Но как исключить первый аргумент `x`, не передавая в него какого-либо значения (даже значения `undefined`), сообщаящего: «Этот аргумент я пропускаю?».

Велико искушение написать `foo(,5)`, но такой синтаксис недопустим. Кажется, нужный эффект может дать вариант `foo.apply(null,[,5])`,

но особенность метода `apply(..)` такова, что в данном случае аргументы будут интерпретироваться как `[undefined, 5]`, то есть все равно не удастся указать на пропуск одного из них.

Дальнейшие эксперименты показывают, что исключить можно только конечные аргументы (расположенные с правой стороны), просто передав их в меньшем количестве, чем ожидается. Опустить аргументы в середине или начале списка нельзя. Это попросту невозможно.

В данном случае к архитектуре JavaScript применяется принцип, который важно запомнить: значение `undefined` трактуется как *отсутствующее*. То есть разницы между `undefined` и *отсутствием* значения нет, по крайней мере, в случае аргументов функций.



По непонятной причине в JS возникают ситуации, когда этот принцип не применим, например в случае с массивами, имеющими пустые слоты. Дополнительную информацию вы найдете в книге *Types & Grammar* серии *You Don't Know JS*.

С учетом всего вышеизложенного теперь мы можем рассмотреть полезный синтаксис, появившийся в ES6 и позволяющий упростить присвоение отсутствующим аргументам значений по умолчанию:

```
function foo(x = 11, y = 31) {  
    console.log( x + y );  
}  
  
foo();                // 42  
foo( 5, 6 );          // 11  
foo( 0, 42 );          // 42  
foo( 5 );              // 36  
  
foo( 5, undefined );  // 36 <-- 'undefined' отсутствуем  
foo( 5, null );        // 5 <-- null приводится к '0'  
  
foo( undefined, 6 );  // 17 <-- 'undefined' отсутствуем  
foo( null, 6 );        // 6 <-- null приводится к '0'
```


Обратите внимание на результаты и на то, как они подчеркивают небольшие отличия и одновременно схожесть с предыдущими подходами.

Выражение `x = 11` в объявлении функции больше напоминает `x !== undefined ? x : 11`, чем распространенная идиома `x || 11`, поэтому в случае присвоения параметру значения по умолчанию следует с осторожностью подходить к преобразованию более ранних версий кода к синтаксису ES6.



Параметр `rest/gather` (см. выше раздел «Операторы Spread и Rest») не может иметь значения по умолчанию. Соответственно, хотя функция `foo(...vals=[1,2,3])` { и кажется интересной возможностью, подобный синтаксис недопустим. При необходимости вам придется добавлять подобные логические схемы вручную.

Выражения как значения по умолчанию

Значения функций по умолчанию не ограничиваются простыми числами, например 31, — эту роль может играть любое допустимое выражение и даже вызов функции:

```
function bar(val) {  
  console.log( "bar called!" );  
  return y + val;  
}  
  
function foo(x = y + 3, z = bar( x )) {  
  console.log( x, z );  
}  
  
var y = 5;  
foo();           // "bar called"  
                // 8 13  
foo( 10 );      // "bar called"  
                // 10 15  
  
y = 6;  
foo( undefined, 10 );  // 9 10
```

Как видите, выражения со значениями по умолчанию вычисляются отложено: это делается только тогда, когда требуется результат — то есть когда возникает ситуация, при которой аргумент параметра отсутствует или имеет значение `undefined`.

Тут есть еще одна тонкость. Формальные параметры в объявлении функции находятся в своей собственной области видимости (ее можно представить как невидимую обертку вокруг скобок `(..)` в объявлении функции), а не в области видимости функции. Это означает, что ссылка на идентификатор в выражении со значением по умолчанию первым делом сопоставляется с областью видимости формальных параметров, а потом уже — с внешней областью видимости. Более подробную информацию вы найдете в книге *Scope & Closures* этой серии.

Рассмотрим пример:

```
var w = 1, z = 2;

function foo( x = w + 1, y = x + 1, z = z + 1 ) {
  console.log( x, y, z );
}

foo();           // ReferenceError
```

Выражение со значением по умолчанию `w + 1` первым делом ищет переменную `w` в области видимости формальных параметров, но, не обнаружив ее там, берет значение `w` извне. Затем для выражения `x + 1` в области видимости формальных параметров ищется переменная `x`, которая, к счастью, уже оказывается инициализированной, так что присвоить значение переменной `y` удастся без проблем.

А вот параметр-переменная `z` в выражении `z + 1` на данный момент пока не инициализирована, поэтому поиск ее во внешней области видимости не производится.

Как упоминалось в разделе «Оператор `let`», в ES6 встречается проблема с временной мертвой зоной, мешающая доступу к неиници-

ализированным переменным. Именно поэтому выражение `z + 1` приводит к ошибке `ReferenceError`.

Выражения со значением по умолчанию можно использовать даже в качестве встроеного вызова функции (в таком случае обычно говорят о немедленно называемых функциях — IIFE), хотя с точки зрения понятности кода это не самая лучшая идея:

```
function foo( x =  
    (function(v){ return v + 11; })( 31 )  
) {  
    console.log( x );  
}  
foo();           // 42
```

Случаи, когда выражения со значением по умолчанию уместны в качестве IIFE (или любых других встроенных исполняемых функциональных выражений) крайне редки. Если у вас возникает искушение использовать их подобным образом, остановитесь и еще раз подумайте!



Если IIFE попытается получить доступ к идентификатору `x`, не объявив предварительно собственный `x`, это также приведет к ошибке TDZ, как и в обсуждавшихся выше случаях.

Выражение со значением по умолчанию в предыдущем фрагменте кода представляет собой IIFE, то есть функцию, исполняемую там, где она располагается, с использованием значения (31). Если отбросить эту часть, значением по умолчанию, присваиваемым переменной `x`, станет ссылка на саму функцию, что можно рассматривать как обратный вызов по умолчанию. В некоторых случаях подобная техника оказывается полезной, например:

```
function ajax(url, cb = function(){} ) {  
    // ..  
}  
ajax( "http://some.url.1" );
```

Здесь нам, по существу, требуется, чтобы значение по умолчанию `cb` представляло собой вызов функции вхолостую, если не указан другой вариант. Функциональное выражение в данном случае — всего лишь ссылка на функцию, а не функция, вызывающая саму себя (в конце отсутствуют скобки `()`, необходимые при вызове), а нам именно это и нужно.

С момента появления JS существует полезное, но малоизвестное свойство `Function.prototype`, представляющее собой пустую, фиктивную функцию. Поэтому можно написать в объявлении `cb = Function.prototype`, избежав создания встроенного функционального выражения.

Деструктурирующее присваивание

В ES6 появилась новая синтаксическая функциональная особенность, которая называется *деструктурирующим присваиванием* (destructuring). Возможно, вам будет проще понять ее суть, если вы взглянете на процесс как на *структурированное присваивание* (structured assignment). Рассмотрим пример:

```
function foo() {  
    return [1,2,3];  
}  
  
var    tmp = foo(),  
      a = tmp[0], b = tmp[1], c = tmp[2];  
  
console.log( a, b, c );           // 1 2 3
```

В данном случае мы вручную присвоили значения в массиве, который функция `foo()` возвращает в отдельные переменные `a`, `b` и `c`, причем для выполнения этой операции нам (к сожалению) потребовалась дополнительная переменная `tmp`.

Эту процедуру можно проделать и с объектами:

```
function bar() {  
  return {  
    x: 4,  
    y: 5,  
    z: 6  
  };  
}  
  
var    tmp = bar(),  
      x = tmp.x, y = tmp.y, z = tmp.z;  
  
console.log( x, y, z );           // 4 5 6
```

Значение свойства `tmp.x` присваивается переменной `x`, а значения свойств `tmp.y` и `tmp.z` — переменным `y` и `z` соответственно.

Выполняемое вручную присваивание значений массива или свойств объектов можно воспринимать как *структурированное присваивание* (structured assignment). В ES6 добавлен специальный синтаксис для *деструктурирующего присваивания*, которое в числе прочего применимо к массивам и объектам. В результате дополнительная переменная `tmp` уже не требуется, что делает код более понятным. Рассмотрим пример:

```
var    [ a, b, c ] = foo();  
var    { x: x, y: y, z: z } = bar();  
  
console.log( a, b, c );           // 1 2 3  
console.log( x, y, z );           // 4 5 6
```

Скорее всего, справа от оператора `=` вы привыкли видеть в качестве присваиваемого значения что-нибудь вроде `[a,b,c]`. Деструктурирующее присваивание зеркально переворачивает этот шаблон, в результате чего запись `[a,b,c]` слева от оператора присваивания трактуется уже как своего рода шаблон для разложения находящегося справа массива на составные части и присвоения полученных значений отдельным переменным.

Аналогичным образом запись `{ x: x, y: y, z: z }` определяет шаблон для присваивания отдельным переменным значений из `bar()`.

Шаблон присваивания свойств объекта

Рассмотрим синтаксис `{ x: x, .. }` из предыдущего фрагмента кода более подробно. Если переменная создается с тем же именем, что и свойство объекта, можно сократить запись:

```
var { x, y, z } = bar();  
console.log( x, y, z );           // 4 5 6
```

Здорово, не так ли?

Но какую часть мы отбрасываем в записи `{ x, .. }` — `x`: или `: x`?

Используя сокращенный синтаксис, мы отбрасываем часть `x:.` Может показаться, что это не имеет особого значения, но сейчас вы поймете всю важность данной детали.

Зачем при наличии короткой формы записи вообще пользоваться длинной? Дело в том, что более длинная версия кода позволяет присвоить значение переменной с другим именем, что в некоторых ситуациях может оказаться крайне полезным:

```
var { x: bam, y: baz, z: bap } = bar();  
console.log( bam, baz, bap );    // 4 5 6  
console.log( x, y, z );          // ReferenceError
```

Нужно понять небольшую, но крайне важную особенность этой формы деструктурирования объекта. Чтобы продемонстрировать подводные камни, о которых вам следует знать, мы рассмотрим шаблон определения обычных объектных литералов:

```
var X = 10, Y = 20;  
  
var o = { a: X, b: Y };  
  
console.log( o.a, o.b );         // 10 20
```

Мы знаем, что в записи `{ a: X, b: Y }` фрагмент `a` представляет собой свойство объекта, а `X` — исходное значение, которое ему присваи-

вается. Другими словами, мы имеем синтаксический шаблон `target: source`, или, что будет более наглядно, `property-alias: value`. Этот шаблон интуитивно понятен, так как аналогичен процедуре присваивания с помощью оператора `=`, которая проходит по шаблону `target = source`.

Но при деструктурирующем присваивании объекта, когда мы помещаем синтаксическую конструкцию `{ .. }`, напоминающую объектный литерал, слева от оператора присваивания `=`, мы инвертируем шаблон `target: source`.

Напомню, как это выглядит:

```
var { x: bam, y: baz, z: bap } = bar();
```

В данном случае используется уже синтаксический шаблон `source: target` (или `value: variable-alias`). Запись `x: bam` означает, что свойство `x` является исходным значением, а `bam` — целевой переменной, которой оно присваивается. Другими словами, объектные литералы создаются как `target <-- source`, в то время как деструктурирующее присваивание объекта происходит по принципу `source --> target`. Видите, как все переворачивается?

На данный синтаксис можно взглянуть и с другой точки зрения, которая, вполне вероятно, покажется вам более понятной. Рассмотрим пример:

```
var    aa = 10, bb = 20;

var o =    { x: aa, y: bb };
var    { x: AA, y: BB } = o;

console.log( AA, BB );           // 10 20
```

В строчке `{ x: aa, y: bb }` `x` и `y` — это свойства объекта, как и в строчке `{ x: AA, y: BB }`.

Помните, выше я писал, что при записи `{ x, .. }` отбрасывается часть `x: ?` В двух рассматриваемых строчках после отбрасывания частей `x:`

и `y`: остаются только `aa`, `bb` и `AA`, `BB`, что по существу — впрочем, чисто теоретически, а не на самом деле — является присваиванием из `aa` в `AA` и из `bb` в `BB`.

Такая симметричность, возможно, позволит вам понять, почему для этой функциональной особенности ES6 синтаксический шаблон был намеренно перевернут.



Лично я предпочел бы для деструктурирующего присваивания синтаксис `{ AA: x , BB: y }`, так как это позволило бы в обоих случаях сохранить более привычный шаблон `target: source`. Но, как ни прискорбно, нам с вами остается только приучить себя пользоваться инвертированной формой.

Не только в объявлениях

До этого момента мы использовали деструктурирующее присваивание только при объявлении переменных с помощью оператора `var` (кстати, с таким же успехом мы могли бы взять оператор `let` или `const`), но разбиение на отдельные элементы уместно при любом присваивании, а не только при том, которое производится в момент объявления.

Рассмотрим пример:

```
var a, b, c, x, y, z;
```

```
[a,b,c] = foo();  
( { x, y, z } = bar() );
```

```
console.log( a, b, c );           // 1 2 3  
console.log( x, y, z );           // 4 5 6
```

Здесь переменные уже объявлены и деструктуризация сопровождается только операцией присваивания.



Если деструктурирующее присваивание объекта происходит не внутри оператора `var/let/const`, все выражение присваивания следует поместить в круглые скобки (), потому что в противном случае находящееся слева содержимое фигурных скобок { .. } будет рассматриваться как блок, а не как объект.

Что замечательно, выражения присваивания (а, у и т. п.) не ограничены идентификаторами переменных. Можно использовать любые допустимые выражения присваивания. Например:

```
var o = {};  
  
[o.a, o.b, o.c] = foo();  
( { x: o.x, y: o.y, z: o.z } = bar() );  
  
console.log( o.a, o.b, o.c );           // 1 2 3  
console.log( o.x, o.y, o.z );           // 4 5 6
```

В деструктурирующем присваивании могут фигурировать даже выражения для вычисляемых свойств. Например:

```
var    which = "x",  
      o = {};  
  
( { [which]: o[which] } = bar() );  
  
console.log( o.x );                     // 4
```

Фрагмент `[which]`: представляет собой вычисляемое свойство, дающее в результате `x`, — свойство, которое будет извлечено из рассматриваемого объекта и взято в качестве присваиваемых данных. Запись `o[which]` соответствует обычной ссылке на ключ объекта, которая равняется `o.x` и выступает как цель присваивания.

Присваивание в общей форме можно использовать для создания отображений/преобразований объекта, например:

```
var    o1 = { a: 1, b: 2, c: 3 },
        o2 = {};

( { a: o2.x, b: o2.y, c: o2.z } = o1 );

console.log( o2.x, o2.y, o2.z );      // 1 2 3
```

Можно преобразовать объект в массив, например так:

```
var    o1 = { a: 1, b: 2, c: 3 },
        a2 = [];

( { a: a2[0], b: a2[1], c: a2[2] } = o1 );

console.log( a2 );                    // [1,2,3]
```

Или сделать наоборот:

```
var    a1 = [ 1, 2, 3 ],
        o2 = {};

[ o2.a, o2.b, o2.c ] = a1;

console.log( o2.a, o2.b, o2.c );      // 1 2 3
```

Можно преобразовать один массив в другой:

```
var
  a1 = [ 1, 2, 3 ],
  a2 = [];

[ a2[2], a2[0], a2[1] ] = a1;

console.log( a2 );                    // [2,3,1]
```

Более того, традиционную задачу по обмену значениями двух переменных можно решать, не прибегая к дополнительной переменной:

```
var    x = 10, y = 20;

[ y, x ] = [ x, y ];

console.log( x, y );                  // 20 10
```



Будьте осторожны: не следует вставлять в объявления операцию присваивания, если только вы не хотите, чтобы как объявления рассматривались все выражения присваивания *одновременно*. В противном случае вы получите синтаксические ошибки. Именно по этой причине в вышеприведенном примере я отделил операцию `var a2 = []` от деструктурирующего присваивания `[a2[0], ..] = ...`. Запись `var [a2[0], ..] = ..` попросту не имеет смысла, так как `a2[0]` — некорректный идентификатор объявления; кроме того, очевидно, что невозможно неявным образом создать объявление `var a2 = []`.

Повторные присваивания

Форма деструктурирующего присваивания объектов позволяет произвольное число раз указывать свойство источника (содержащее значение любого типа). Например:

```
var { a: X, a: Y } = { a: 1 };
```

```
X;           // 1  
Y;           // 1
```

Это означает, что вы можете деструктурировать свойства, которые представляют собой подобъекты/массивы, и фиксировать значение подобъекта/массива. Например:

```
var { a: { x: X, x: Y }, a } = { a: { x: 1 } };
```

```
X;           // 1  
Y;           // 1  
a;           // { x: 1 }
```

```
( { a: X, a: Y, a: [ Z ] } = { a: [ 1 ] } );
```

```
X.push( 2 );  
Y[0] = 10;
```

```
X;           // [10, 2]  
Y;           // [10, 2]  
Z;           // 1
```

Не стоит располагать операции деструктурирующего присваивания на одной строчке, как это было сделано в приведенных ранее примерах. Ради удобства чтения лучше выделять каждой операции отдельную строку с подходящим отступом, как в формате JSON или при использовании значений, заданных объектами-литералами.

```
// сложнее читается:
var    { a: { b: [ c, d ], e: { f } }, g } = obj;

// лучший вариант:
var    {
  a: {
    b: [ c, d ],
    e: { f }
  },
  g
} = obj;
```

Помните, что деструктурирующее присваивание используется не только для уменьшения количества набираемых на клавиатуре символов, но и для улучшения читабельности кода.

Выражения деструктурирующего присваивания

Результатом выражения присваивания, в процессе которого происходит извлечение данных из объекта или массива, становится появление в правой части его значений. Например:

```
var    o = { a:1, b:2, c:3 },
        a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );    // 1 2 3
p === o;                  // true
```

В этом фрагменте кода переменной `p` присваивается ссылка на объект `o`, а не одно из значений `a`, `b` или `c`. Аналогичная ситуация наблюдается при извлечении данных из массива:

```
var    o = [1,2,3],
      a, b, c, p;

p = { a, b, c } = o;

console.log( a, b, c );    // 1 2 3
p === o;                   // true
```

Выполняя сквозной перенос итоговых значений объекта/массива, можно соединить выражения деструктурирующего присваивания в цепочку:

```
var    o = { a:1, b:2, c:3 },
      p = [4,5,6],
      a, b, c, x, y, z;

( {a} = {b,c} = o );
[x,y] = [z] = p;

console.log( a, b, c );    // 1 2 3
console.log( x, y, z );    // 4 5 4
```

Слишком много, слишком мало, в самый раз

При деструктурирующем присваивании объектов и массивов вовсе не обязательно присваивать все имеющиеся значения. Например:

```
var [,b] = foo();
var { x, z } = bar();

console.log( b, x, z );    // 2 4 6
```

Отброшены значения 1 и 3 из массива `foo()` и значение 5 из объекта `bar()`.

Если попытаться присвоить большее количество значений, чем появляется в процессе деструктуризации, лишние переменные совершенно логично получают значение `undefined`:

```
var [,c,d] = foo();  
var { w, z } = bar();  
  
console.log( c, z );    // 3 6  
console.log( d, w );    // undefined undefined
```

Подобное поведение следует ранее установленному принципу «значение `undefined` трактуется как отсутствующее».

Выше мы рассмотрели оператор `...`, который позволяет как разбивать массивы на отдельные значения, так и объединять наборы значений в массивы. Такое поведение он демонстрирует не только в объявлениях функций, но и при деструктурирующем присваивании. Для иллюстрации возьмем уже встречавшийся ранее фрагмент кода:

```
var a = [2,3,4];  
var b = [ 1, ...a, 5 ];  
  
console.log( b );      // [1,2,3,4,5]
```

Здесь мы видим, что запись `...a` приводит к извлечению значений, так как мы имеем дело с массивом `[...]`. При этом вставка в деструктурирующее присваивание массива записи `...a` приведет к сбору значений:

```
var a = [2,3,4];  
var [ b, ...c ] = a;  
  
console.log( b, c );    // 2 [3,4]
```

Деструктурирующее присваивание `var [..] = a` распределяет значения из `a` в соответствии с заданным внутри квадратных скобок `[..]` шаблоном. Сначала первое значение из массива `a` (2) сохра-

няется под именем `b`, после чего оператор `...` собирает остальные значения (3 и 4) в массив с именем `c`.



Мы знаем, как оператор `...` работает с массивами, но что он делает с объектами? В ES6 эта функциональная особенность отсутствует, но в главе 8 вы найдете обсуждение подхода, выходящего за пределы ES6, когда `...` применяется для разделения или сбора объектов.

Присваивание значений по умолчанию

Обе формы деструктуризации дают нам возможность присваивать значения по умолчанию. При этом используется синтаксис, знакомый вам по рассмотренной выше процедуре присваивания постоянных значений аргументам функций. Смотрите сами:

```
var [ a = 3, b = 6, c = 9, d = 12 ] = foo();  
var { x = 5, y = 10, z = 15, w = 20 } = bar();  
  
console.log( a, b, c, d );    // 1 2 3 12  
console.log( x, y, z, w );    // 4 5 6 20
```

Присваивание значений по умолчанию можно скомбинировать с другими вариантами присваивания, которые мы разбирали ранее. Например:

```
var { x, y, z, w: WW = 20 } = bar();  
  
console.log( x, y, z, WW );    // 4 5 6 20
```

Использовать объекты или массивы в качестве значений по умолчанию при деструктуризации не стоит, так как в результате можно получить код, понять который будет крайне непросто:

```
var x = 200, y = 300, z = 100;  
var o1 = { x: { y: 42 }, z: { y: z } };  
  
( { y: x = { y: y } } = o1 );  
( { z: y = { y: z } } = o1 );  
( { x: z = { y: x } } = o1 );
```

Можете ли вы, глядя на этот фрагмент кода, назвать конечные значения `x`, `y` и `z`? Подозреваю, что вы надолго задумались. Я помогу вам и напишу верный ответ:

```
console.log( x.y, y.y, z.y );    // 300 100 42
```

Надеюсь, мне удалось наглядно убедить вас, что деструктурирующее присваивание — это инструмент не только крайне полезный, но еще острый, при неблагоразумном использовании способный нанести вред.

Вложенное деструктурирующее присваивание

Деструктурирующее присваивание возможно при любом уровне вложенности объектов или массивов:

```
var a1 = [ 1, [2, 3, 4], 5 ];
var o1 = { x: { y: { z: 6 } } };

var [ a, [ b, c, d ], e ] = a1;
var { x: { y: { z: w } } } = o1;

console.log( a, b, c, d, e );    // 1 2 3 4 5
console.log( w );                // 6
```

Вложенная деструктуризация может оказаться простым способом сведения воедино пространств имен объектов. Например:

```
var App = {
  model: {
    User: function(){ .. }
  }
};

// вместо:
// var User = App.model.User;

var { model: { User } } = App;
```


Деструктуризация параметров

Можете ли вы показать, где в следующем фрагменте кода делается присваивание?

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

```
foo( 42 );
```

В данном случае эта операция выполняется неявно: 42 (аргумент) присваивается *x* (параметру) во время выполнения `foo(42)`. Но если присваиванием является пара параметр/аргумент, логично предположить, что оно может быть деструктурирующим, не так ли? Именно так!

Рассмотрим деструктуризацию массива для параметров:

```
function foo( [ x, y ] ) {  
    console.log( x, y );  
}  
  
foo( [ 1, 2 ] );           // 1 2  
foo( [ 1 ] );             // 1 undefined  
foo( [] );                // undefined undefined
```

Возможна и аналогичная деструктуризация объекта:

```
function foo( { x, y } ) {  
    console.log( x, y );  
}  
  
foo( { y: 1, x: 2 } );     // 2 1  
foo( { y: 42 } );         // undefined 42  
foo( {} );                // undefined undefined
```

Эта техника представляет собой примерную реализацию именованных аргументов (давно необходимой в JS функциональной особенности), поскольку свойства объекта сопоставляются деструк-

турированным параметрам, имеющим те же самые имена. Кроме того, мы заодно получаем необязательные параметры (в любой позиции); вы уже видели, как нужный результат достигался удалением `x`.

Разумеется, в случае деструктуризации параметров нам доступны все ранее обсуждавшиеся варианты процедуры, в том числе связанные с вложенностью, значениями по умолчанию и пр. Этот процесс хорошо комбинируется и с другими возможностями параметров функций в ES6, такими как значения параметров по умолчанию и `rest/gather`-параметры.

Рассмотрим краткие примеры (разумеется, демонстрирующие далеко не все возможные варианты):

```
function f1([ x=2, y=3, z ]) { .. }
function f2([ x, y, ...z], w) { .. }
function f3([ x, y, ...z], ...w) { .. }

function f4({ x: X, y }) { .. }
function f5({ x: X = 10, y = 20 }) { .. }
function f6({ x = 10 } = {}, { y } = { y: 10 }) { .. }
```

Для иллюстрации давайте проанализируем один пример из приведенного фрагмента:

```
function f3([ x, y, ...z], ...w) {
  console.log( x, y, z, w );
}

f3( [] );           // undefined undefined [] []
f3( [1,2,3,4], 5, 6 ); // 1 2 [3,4] [5,6]
```

В данном случае используются два оператора `...`, собирающие значения в массивы (`z` и `w`), причем `...z` собирает значения, оставшиеся в первом массиве-аргументе, в то время как `...w` собирает основные аргументы, оставшиеся после отбрасывания первого.

Значения по умолчанию для деструктуризации и для параметров

Есть один тонкий нюанс, заметить который не так-то просто. Это разница в поведении между значениями по умолчанию деструктуризации и параметра функции. Например:

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {  
    console.log( x, y );  
}  
  
f6();           // 10 10
```

На первый взгляд кажется, что мы объявили для обоих параметров `x` и `y` значение по умолчанию `10`, просто разными способами. Но между этими двумя подходами есть небольшая разница, так как в определенных случаях мы будем наблюдать разное поведение. Рассмотрим пример:

```
f6( {}, {} );    // 10 undefined
```

Вы поняли, почему получен именно такой результат? Очевидно, что именованный параметр `x`, если его не передают как свойство с таким же именем в объект первого аргумента, получает значение по умолчанию, равное `10`.

Но почему у параметра `y` оказывается значение `undefined`? Дело в том, что запись `{ y: 10 }` означает объект как значение по умолчанию параметра функции, а не как значение по умолчанию деструктуризации и поэтому применимо только в случае, когда второй аргумент вообще не передается или передается как `undefined`.

В предыдущем фрагменте кода мы *передаем* второй аргумент `{}`, так что значение по умолчанию `{ y: 10 }` не используется и деструктуризация `{ y }` возникает как функция значения переданного пустого объекта `{}`.

Теперь сравним `{ y } = { y: 10 }` и `{ x = 10 } = { }`.

Что касается варианта с переменной `x`, то, если аргумент первой функции опущен или имеет значение `undefined`, применяется значение по умолчанию пустого объекта `{ }`. Затем вне зависимости от того, какое значение оказалось в положении первого аргумента — это может быть или значение по умолчанию `{ }`, или значение, которое передали вы, — выполняется его деструктуризация с учетом того, что `{ x = 10 }`. При этом ищется свойство `x`. Если обнаружить его не удастся (или оно имеет значение `undefined`), именованный параметр `x` получает значение по умолчанию `10`.

Сделайте глубокий вдох. Несколько раз перечитайте последние абзацы. Чтобы было понятнее, вот вам еще пример кода:

```
function f6({ x = 10 } = {}, { y } = { y: 10 }) {  
  console.log( x, y );  
}  
  
f6(); // 10 10  
f6( undefined, undefined ); // 10 10  
f6( {}, undefined ); // 10 10  
  
f6( {}, {} ); // 10 undefined  
f6( undefined, {} ); // 10 undefined  
  
f6( { x: 2 }, { y: 3 } ); // 2 3
```

Складывается впечатление, что заданное по умолчанию поведение параметра `x` более желательно и адекватно, чем поведение параметра `y`. Именно поэтому важно понять, почему и каким образом различаются формы `{ x = 10 } = { }` и `{ y } = { y: 10 }`.

Если четкого понимания у вас пока не появилось, еще раз перечитайте этот раздел и как следует поэкспериментируйте. Время, которое вы сейчас потратите на то, чтобы уяснить данный крайне тонкий нюанс, с лихвой окупится в будущем.

Вложенные значения по умолчанию: деструктурированные и реструктурированные

Хотя поначалу это не очевидно, но задание значений по умолчанию для свойств вложенного объекта порождает интересную идиому: использование деструктуризации объекта вместе с процессом, который я назвал бы *реструктуризацией* (restructuring).

Рассмотрим набор значений по умолчанию в структуре вложенного объекта:

```
// взято с сайта:  
// http://es-discourse.com/t/partial-default-arguments/120/7
```

```
var    defaults = {  
  options: {  
    remove: true,  
    enable: false,  
    instance: {}  
  },  
  log: {  
    warn: true,  
    error: true  
  }  
};
```

Теперь предположим, что у нас есть объект `config`, в котором фигурируют какие-то из этих значений (но, скорее всего, не все). Предположим также, что вы хотите установить все значения по умолчанию в те места, где их нет, не переопределяя уже существующих специализированных настроек.

```
var    config = {  
  options: {  
    remove: false,  
    instance: null  
  }  
};
```

Разумеется, задачу можно решить вручную, как, вероятно, вы делали в прошлом.

```
config.options = config.options || {};  
config.options.remove = (config.options.remove !== undefined) ?  
    config.options.remove : defaults.options.remove;  
config.options.enable = (config.options.enable !== undefined) ?  
    config.options.enable : defaults.options.enable;  
...
```

Этот код выглядит отвратительно.

Кто-то, возможно, предпочтет подход с присваиванием-перезаписью. У кого-то возникнет соблазн воспользоваться появившимся в ES6 методом `Object.assign(..)` (см. главу 6), чтобы сначала клонировать свойства из значений по умолчанию, а затем выполнить перезапись клонированных свойств из объекта `config`:

```
config = Object.assign( {}, defaults, config );
```

Куда более привлекательный вариант, не так ли? Но есть большая проблема: метод `Object.assign(..)` достаточно поверхностный, при копировании `defaults.options` он формирует только ссылку на объект, не дублируя его свойства в объект `config.options`. Для достижения нужного нам результата метод `Object.assign(..)` пришлось бы применять (можно сказать, рекурсивно) на всех уровнях дерева объекта.



Многие служебные библиотеки/фреймворки для JS предоставляют возможности для глубокого клонирования объектов, но рассмотрение этих инструментов и их подводных камней выходит за рамки нашего обсуждения.

Итак, давайте посмотрим, может ли нам хоть как-то помочь де-структуризация объекта с параметрами по умолчанию:

```
config.options = config.options || {};  
config.log = config.log || {};
```

```
{
  options: {
    remove: config.options.remove = default.options.remove,
    enable: config.options.enable = default.options.enable,
    instance: config.options.instance =
      default.options.instance
  } = {},
  log: {
    warn: config.log.warn = default.log.warn,
    error: config.log.error = default.log.error
  } = {}
} = config;
```

Не так красиво, как обманчивые обещания метода `Object.assign(...)`, но, на мой взгляд, несколько лучше, чем работа вручную. Впрочем, код получился слишком объемным и содержит повторы.

Подход, продемонстрированный в предыдущем фрагменте, оказался эффективным, потому что я принудительно заставил механизм обработки деструктуризации и значений по умолчанию проверить, что свойство `=== undefined`, и принять вместо меня решения о присваивании. Хитрость состоит в том, что я выполняю деструктуризацию объекта `config` (в конце фрагмента вы видите `= config`), а потом повторно присваиваю все полученные при этом значения объекту `config` со ссылками на присваивание `config.options.enable`.

Впрочем, код слишком объемен. Посмотрим, нельзя ли его как-нибудь улучшить.

Если вы знаете, что все свойства, которые вы деструктуризуете, имеют уникальные имена, лучше всего подойдет рассматриваемый ниже трюк. Он применим и в других случаях, но тогда вы получите уже не такой красивый код — придется выполнять деструктуризацию в несколько этапов или же создать уникальные локальные переменные в качестве временных псевдонимов.

Если полностью деструктуризовать все свойства переменных верхнего уровня, их можно будет немедленно реструктуризовать, восстановив исходную структуру вложенного объекта. Но тогда все

эти появившиеся в коде временные переменные заполняют область видимости. Так что мы воспользуемся областью видимости на уровне блока (см. раздел «Объявления на уровне блоков кода» в начале этой главы), создав общий блок `{ }`, включающий в себя весь код:

```
// сливаем 'defaults' в 'config'
{
  // деструктуризация (с присваиваниями значений по умолчанию)
  let {
    options: {
      remove = defaults.options.remove,
      enable = defaults.options.enable,
      instance = defaults.options.instance
    } = {},
    log: {
      warn = defaults.log.warn,
      error = defaults.log.error
    } = {}
  } = config;

  // реструктуризация
  config = {
    options: { remove, enable, instance },
    log: { warn, error }
  };
}
```

Такой код выглядит намного лучше, не так ли?



Вместо общего блока `{ }` и объявления с помощью оператора `let` область видимости можно было бы задать стрелочной IIFE. В этом случае деструктурирующие присваивания/значения по умолчанию оказались бы в списке параметров, а реструктуризацию пришлось бы поместить в оператор `return` в теле функции.

Синтаксис `{ warn, error }` во фрагменте с реструктуризацией, возможно, вам пока незнаком. Это называется краткой формой записи свойств, мы рассмотрим ее в следующем разделе.

Расширения объектных литералов

Стандарт ES6 добавил ряд важных расширений для удобства работы со скромными объектными литералами { .. }.

Краткие свойства

Вы, без сомнения, знакомы со следующей формой объявления объектных литералов:

```
var    x =    2, y = 3,  
      o =    {  
              x: x,  
              y: y  
      };
```

Если вам всегда казалось излишним постоянное повторение `x: x`, у меня для вас хорошие новости. Теперь определение свойства, имя которого совпадает с лексическим идентификатором, можно сократить с `x: x` до `x`. Смотрите:

```
var    x =    2, y = 3,  
      o =    {  
              x,  
              y  
      };
```

Краткие методы

Аналогично только что рассмотренным нами кратким свойствам, функции, присоединенные к свойствам внутри объектного литерала, также получили краткую удобную форму.

Старый способ выглядит так:

```
var    o = {  
      x: function(){  
        // ..  
      },  
      y: function(){  
        // ..  
      }  
    }
```

А реализация в ES6 — так:

```
var    o = {  
      x() {  
        // ..  
      },  
      y() {  
        // ..  
      }  
    }
```



Несмотря на то что запись `x() { .. }` выглядит как обычное сокращение для `x: function() { .. }`, краткие методы обладают особыми вариантами поведения, отсутствующими у их более старых эквивалентов; в частности, допускается применение ключевого слова `super` (см. ниже раздел «Ключевое слово `super`»).

У генераторов (см. главу 4) также есть сокращенная форма метода:

```
var    o = {  
      *foo() { .. }  
    };  
};
```

Краткие и анонимные

Несмотря на всю привлекательность введенных для нашего удобства сокращений, здесь есть небольшой подводный камень, и о нем вам следует знать. Для иллюстрации рассмотрим старую версию кода, которую можно будет попробовать переписать с использованием кратких методов:

```
function runSomething(o) {
    var    x = Math.random(),
          y = Math.random();

    return o.something( x, y );
}

runSomething( {
    something: function something(x,y) {
        if (x > y) {
            // рекурсивный вызов с заменой 'x' и 'y'
            return something( y, x );
        }

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

Этот несерьезный фрагмент кода генерирует два случайных числа и вычитает меньшее из большего. Но здесь важно не то, что он делает, а определения. Сфокусируемся на определениях объектного литерала и функции:

```
runSomething( {
    something: function something(x,y) {
        // ..
    }
} );
```

Почему мы одновременно пишем `something`: и `function something`? Не является ли это избыточным повторением? Разумеется, нет, эти записи нужны для разных целей. Свойство `something` представляет собой способ вызова `o.something(..)`, своего рода открытое имя. Во втором же случае `something` — это лексическое имя для ссылки на функцию из нее самой, позволяющее выполнить рекурсию.

Вы понимаете, зачем в строчке `return something(y,x)` требуется имя `something` для ссылки на эту функцию? У нас нет лексического имени для объекта, такого, чтобы можно было написать `return o.something(y,x)` или нечто подобное.

На самом деле идентификационные имена у объектных литералов встречаются довольно часто. Например:

```
var controller = {  
  makeRequest: function(..){  
    // ..  
    controller.makeRequest(..);  
  }  
};
```

Хорошо ли так делать? Скорее всего, нет. Вам кажется, что имя `controller` всегда будет указывать на рассматриваемый объект. Но это предположение может оказаться неверным — функция `makeRequest(..)` не контролирует внешний код и потому не способна гарантировать такое положение вещей. В будущем это может выйти вам боком.

Некоторые программисты предпочитают пользоваться ключевым словом `this`:

```
var controller = {  
  makeRequest: function(..){  
    // ..  
    this.makeRequest(..);  
  }  
};
```

Это выглядит приемлемо и будет работать, если всегда вызывать метод в форме `controller.makeRequest(..)`. Но вы натолкнетесь на проблему со связыванием, попытавшись сделать, к примеру, вот такую вещь:

```
btn.addEventListener( "click", controller.makeRequest, false );
```

Разумеется, проблему можно попытаться решить, передав в качестве ссылки на обработчик `controller.makeRequest.bind(controller)`. Но это, увы, не очень привлекательный вариант.

К примеру, что вы будете делать, если внутренний вызов `this.makeRequest(..)` следует совершить из вложенной функции? Снова возникает опасность реализации связывания с помощью ключево-

го слова `this`, причем зачастую все решается топорным объявлением `var self = this`, как в этом примере:

```
var controller = {
  makeRequest: function(..){
    var self = this;
    btn.addEventListener( "click", function(){
      // ..
      self.makeRequest(..);
    }, false );
  }
};
```

Этот код еще хуже.



Дополнительную информацию о правилах и особенностях связывания вы найдете в главах 1–2 книги *this & Object Prototypes* этой серии.

Хорошо, но какое отношение все вышесказанное имеет к кратким методам? Вспомним определение нашего метода `something(..)`:

```
runSomething( {
  something: function something(x,y) {
    // ..
  }
} );
```

Второе `something` в данном случае обеспечивает весьма удобный лексический идентификатор, всегда указывающий на саму функцию и идеально подходящий для рекурсии, связывания/открепления обработчиков событий и тому подобных вещей. Если использовать его, исчезает необходимость прибегать к ключевому слову `this` или пользоваться ненадежными ссылками на объект.

Великолепно!

А сейчас мы попробуем переписать ссылку на функцию с использованием краткой формы метода из ES6:

```
runSomething( {  
  something(x,y) {  
    if (x > y) {  
      return something( y, x );  
    }  
  
    return y - x;  
  }  
} );
```

На первый взгляд все замечательно, но выполнение этого кода будет прервано. Вызов `return something(...)` не обнаружит идентификатора `something`, что приведет к ошибке `ReferenceError`. В чем причина?

Приведенный выше фрагмент кода в стандарте ES6 интерпретируется следующим образом:

```
runSomething( {  
  something: function(x,y){  
    if (x > y) {  
      return something( y, x );  
    }  
  
    return y - x;  
  }  
} );
```

Посмотрите внимательно. Видите, где проблема? Определение краткого метода подразумевает `something: function(x,y)`. Вы поняли, что здесь пропущено второе `something`, на наличие которого мы полагались? Другими словами, краткие методы подразумевают анонимные функциональные выражения.

Да, дело дрянь.



Возможно, у вас возникла мысль, что хорошим решением в данном случае послужат стрелочные функции `=>`, но на самом деле они также представляют собой анонимные функциональные выражения и потому неэффективны. О них мы подробно поговорим в разделе «Стрелочные функции» этой главы.

Частично улучшает ситуацию тот факт, что наш краткий метод `something(x,y)` полностью анонимным не будет. В разделе «Имена функций» главы 7 вы найдете информацию о правилах вывода имен функций. Эти правила не помогут организовать рекурсию, но, по крайней мере, пригодятся в процессе отладки.

Какой же вывод следует сделать относительно кратких методов? Они лаконичны и удобны, но пользоваться ими можно, только если вы не собираетесь выполнять рекурсию или передавать функцию в обработчики событий. В противном случае вам на помощь придет старый вариант определения методов `something: function something(..)`.

Впрочем, для большинства методов прекрасно подойдет краткая форма определения, что не может не радовать! Главное, будьте осторожны в тех немногих случаях, когда анонимность способна принести вред.

Методы чтения/записи из ES5

Технически в стандарте ES5 были определены литеральные формы методов чтения/записи, но ими практически никто не пользовался из-за малого числа транскомпиляторов, способных обработать новый синтаксис (кстати, единственный новый синтаксис, появившийся в ES5). Потому, хотя эту функциональную особенность нельзя считать новинкой стандарта ES6, мы кратко напомним ее суть, так как по мере развития языка растет вероятность, что она станет применяться шире.

Рассмотрим пример:

```
var    o = {  
  __id: 10,  
  get id() { return this.__id++; },  
  set id(v) { this.__id = v; }  
}  
  
o.id;           // 10
```

```

o.id;           // 11
o.id = 20;
o.id;           // 20

// ...u:
o.__id;         // 21
o.__id;         // 21 – все еще!

```

Эти литеральные формы методов чтения и записи присутствуют и в классах, о которых мы поговорим в главе 3.



Наверное, это не очевидно, но у литерала метода записи должен быть всего один объявленный параметр; его пропуск или добавление других параметров недопустимы. Для него возможна деструктуризация и значения по умолчанию (например, `set id({ id: v = 0 }) { .. }`), но он не способен выступать `gather/rest`-параметром (`set id(...v) { .. }`).

Имена вычисляемых свойств

Вероятно, вам приходилось сталкиваться с ситуацией, представленной в следующем фрагменте кода, когда у вас есть одно или несколько имен свойств, которые взяты из какого-либо выражения и потому не могут быть помещены в объектный литерал.

```

var    prefix = "user_";

var    o = {
  baz: function(..){ .. }
};

o[ prefix + "foo" ] = function(..){ .. };
o[ prefix + "bar" ] = function(..){ .. };
..

```

В ES6 к определению объектного литерала добавлен синтаксис, позволяющий указывать выражение, которое следует вычислить и результат которого является назначенным именем свойства. Рассмотрим пример:


```
var    prefix = "user_";

var    o = {
  baz: function(..){ .. },
  [ prefix + "foo" ]: function(..){ .. },
  [ prefix + "bar" ]: function(..){ .. }
  ..
};
```

Внутри скобок [..], которые в определении объектного литерала располагаются на месте имени свойства, можно поместить любое действительное выражение.

Вероятно, шире всего имена вычисляемых свойств будут использоваться в данных типа Symbols (о них мы поговорим в соответствующем разделе этой главы), например так:

```
var    o = {
  [Symbol.toStringTag]: "really cool thing",
  ..
};
```

В данном случае `Symbol.toStringTag` — это специальное встроенное значение, которое мы получаем с помощью синтаксиса [..], что позволяет нам присвоить имени специального свойства значение "really cool thing".

Также имена вычисляемых свойств могут появляться как имена кратких методов и кратких генераторов:

```
var    o = {
  ["f" + "oo"]() { .. }    // вычисляемый краткий метод
  *["b" + "ar"]() { .. }  // вычисляемый краткий генератор
};
```

Установка [[Prototype]]

Я не буду рассматривать прототипы детально, так что, если вам требуется дополнительная информация, воспользуйтесь книгой *this & Object Prototypes* этой серии.

Иногда полезно установить `[[Prototype]]` объекта одновременно с объявлением его объектного литерала. Изначально для многих JS-движков это расширение было нестандартным, но в ES6 его стандартизовали:

```
var    o1 = {  
    // ..  
};  
var    o2 = {  
    __proto__: o1,  
    // ..  
};
```

Переменная `o2` объявлена с помощью обычного объектного литерала, но одновременно она представляет собой `[[Prototype]]`-, связанный с переменной `o1`. Обратите внимание: в данном случае имя свойства `__proto__` может быть одновременно строкой `"__proto__"`, но *не* именем, полученным как результат вычисляемого свойства (см. предыдущий раздел).

Свойство `__proto__` неоднозначно, если не сказать больше. Это существующее не одно десятилетие проприетарное расширение JS, которое наконец не без колебаний стандартизовали в ES6. Многие разработчики считают, что его вообще не следует использовать. Оно упоминается в приложении В спецификации ES6, то есть в разделе, где перечислены вещи, которые требовалось стандартизировать исключительно для обеспечения совместимости.



Отчасти я одобряю использование свойства `__proto__` в качестве ключа в определении объектного литерала, но ни в коем случае не рекомендую применять его как свойство объекта, например, `o.__proto__`. Такая форма одновременно является методом чтения и методом записи (опять же, это было сделано для обеспечения совместимости), но есть и куда более приемлемые варианты. Подробнее данная тема рассматривается в книге *this & Object Prototypes* этой серии.

Для задания `[[Prototype]]` существующего объекта в ES6 применяется метод `Object.setPrototypeOf(..)`. Рассмотрим пример:

```
var    o1 = {  
    // ..  
};  
  
var    o2 = {  
    // ..  
};  
  
Object.setPrototypeOf( o2, o1 );
```



Объекты мы будем обсуждать в главе 6. В разделе «Статическая функция `Object.setPrototypeOf(..)`» вы найдете дополнительные сведения о методе `Object.setPrototypeOf(..)`, а в разделе «Статическая функция `Object.assign(..)`» — форму сопоставления переменных `o2` и `o1` посредством прототипов.

Ключевое слово `super`

Ключевое слово `super`, как правило, рассматривается в связи с классами. Но благодаря тому, что в JS понятие класса приравнивается к понятию объекта с прототипом, ключевое слово `super` эффективно работает с краткими методами обычных объектов.

Рассмотрим пример:

```
var    o1 = {  
    foo() {  
        console.log( "o1:foo" );  
    }  
};  
  
var    o2 = {  
    foo() {  
        super.foo();  
        console.log( "o2:foo" );  
    }  
}
```

```
};  
  
Object.setPrototypeOf( o2, o1 );  
  
o2.foo();           // o1:foo  
                    // o2:foo
```



Ключевое слово `super` допустимо только в сокращенных методах, а не в свойствах регулярных функциональных выражений. Кроме того, оно действительно только в форме `super.XXX` (для доступа к свойству/методу) и не применяется в форме `super()`.

Ссылка `super` в методе `o2.foo()` статически привязана к `o2`, а конкретнее к `[[Prototype]]` переменной `o2`. В данном случае ключевое слово `super`, по сути, означает `Object.getPrototypeOf(o2)` — что в итоге сводится к `o1`, именно так обнаруживается и вызывается метод `o1.foo()`.

Детально ключевое слово `super` рассматривается в разделе «Классы» главы 3.

Шаблонные строки

В начале этого раздела мне придется дать имя функциональной особенности ES6, которая зачастую понимается по-разному, в зависимости от того, в каком контексте человеку доводилось работать с *шаблонами*.

Для многих разработчиков этот термин связан прежде всего с многообразными отрисовываемыми фрагментами текста, предоставляемыми разными шаблонизаторами (Mustache, Handlebars и т. п.). В ES6 он имеет примерно такое же значение. Скажем, он обозначает способ объявления встроенных шаблонных строк, допускающих повторное отображение. Но этим данная функциональная особенность не ограничивается.

Так что прежде чем мы двинемся вперед, я дам ей более подходящее название: *интерполированные строковые литералы* (interpolated string literals).

Вы уже хорошо осведомлены об объявлении строковых литералов с помощью разделителя " или ', и знаете, что это не *умные строки* (smart strings), которые присутствуют в некоторых языках и чье содержимое анализируется на предмет выражений интерполяции.

Но в ES6 появился новый тип строкового литерала, использующий в качестве разделителя знак обратной кавычки '. Такие строковые литералы позволяют встраивать базовые строковые выражения интерполяции, которые затем автоматически анализируются и вычисляются.

Вот как это выглядело до появления ES6:

```
var name = "Kyle";

var greeting = "Hello " + name + "!";

console.log( greeting );           // "Hello Kyle!"
console.log( typeof greeting );    // "string"
```

Теперь посмотрим на вариант кода в ES6:

```
var name = "Kyle";

var greeting = `Hello ${name}!`;

console.log( greeting );           // "Hello Kyle!"
console.log( typeof greeting );    // "string"
```

Как видите, мы заключили в обратные кавычки '`' набор символов, который интерпретируется как строковый литерал, причем все выражения вида `\${..}` немедленно анализируются и вычисляются — эти действия называются замысловатым термином *интерполяция* (interpolation), намного более точным, чем *обработка по шаблону* (templating).

Результатом такого интерполированного выражения для строкового литерала оказывается старая добрая строка, присваиваемая переменной `greeting`.



Запись `typeof greeting == "string"` показывает, почему нельзя воспринимать эти сущности как специальные шаблонные значения: вы не можете присвоить чему-нибудь еще не вычисленный литерал и повторно его использовать. Строковый литерал `'..'` больше напоминает IIFE, потому что он тоже вычисляется в процессе выполнения. Результатом вычисления становится обычная строка.

Одно из преимуществ интерполированных строковых литералов — возможность разбивать их на отдельные строки:

```
var text =  
'Now is the time for all good men  
to come to the aid of their  
country!';  
  
console.log( text );  
// Now is the time for all good men  
// to come to the aid of their  
// country!
```

Переносы строк в интерполированных строковых литералах фиксируются в строковом значении.

Если в значении литерала отсутствует явная управляющая последовательность, значение символа возврата каретки `\r` (кодировка точка U+000D) или возврата каретки плюс символа новой строки `\r\n` (кодировка точки U+000D и U+000A) приводится к виду `a \n`, то есть к символу новой строки (кодировка точка U+000A). Впрочем, у вас не должно быть причин для беспокойства: подобное приведение происходит редко и, скорее всего, вы столкнетесь с ним только при копировании текста и его вставке в ваш JS-файл.

Интерполированные выражения

Внутри интерполированных строковых литералов `${..}` могут находиться любые допустимые выражения, в том числе вызовы функций, вызовы встроенных функциональных выражений и даже другие интерполированные строковые литералы.

Рассмотрим пример:

```
function upper(s) {
    return s.toUpperCase();
}

var who = "reader";

var text =
'A very ${upper( "warm" )} welcome
to all of you ${upper( '${who}s' )}!';

console.log( text );
// A very WARM welcome
// to all of you READERS!
```

В данном случае внутренний интерполированный строковый литерал `'${who}s'` становится более удобным после объединения переменной `who` со строкой `"s"`, в противовес выражению `who + "s"`. Бывают ситуации, когда целесообразно использовать вложенный интерполированный строковый литерал, но не стоит прибегать к подобным вещам слишком часто, равно как и создавать слишком глубокие уровни вложенности.

В подобных случаях велики шансы, что при работе со строковым значением придется повысить уровень абстракции.



Хотелось бы напомнить вам о важности такой вещи, как читабельность кода. Тот факт, что вы *можете* что-то сделать, не означает, что вы *должны* это делать. Старайтесь не перебарщивать с новыми элементами из ES6, иначе ваш код окажется слишком сложным для вашего восприятия и для восприятия других членов вашей рабочей группы.

Область видимости выражений

Небольшое замечание об области видимости, которая используется при определении переменных в выражениях. Я уже упоминал, что интерполированные строковые литералы напоминают IIFE. Эта аналогия помогает разобраться и с поведением области видимости.

Рассмотрим пример:

```
function foo(str) {  
    var name = "foo";  
    console.log( str );  
}  
  
function bar() {  
    var name = "bar";  
    foo( 'Hello from ${name}!' );  
}  
  
var name = "global";  
  
bar();                // "Hello from bar!"
```

Здесь строковый литерал `'..'` помещен внутрь функции `bar()`, и доступная ему область видимости находит принадлежащую этой функции переменную `name` со значением `"bar"`. Не имеют значения ни глобальная переменная `name`, ни переменная с таким же именем, расположенная внутри функции `foo(..)`. Другими словами, интерполированный строковый литерал попадает в лексический контекст там, где он появился, и никаким образом не может быть связан с динамической областью видимости.

Тегированные шаблонные строки

И снова я вынужден дать функциональной особенности более подходящее название: *тегированные строковые литералы* (tagged string literals).

Честно скажу, что это одна из самых потрясающих новинок, предлагаемых стандартом ES6. На первый взгляд она может показаться странной и не очень полезной. Но после того как вы некоторое время с ней поэкспериментируете, она удивит вас своей практической ценностью.

Например:

```
function foo(strings, ...values) {
  console.log( strings );
  console.log( values );
}

var desc = "awesome";

foo'Everything is ${desc}!';
// [ "Everything is ", "!" ]
// [ "awesome" ]
```

Посмотрим внимательно, что происходит в этом фрагменте кода. Во-первых, в глаза бросается элемент `foo'Everything...'`; . Вряд ли раньше вам доводилось видеть подобное. Что же это такое?

По сути, это особый вид вызова функции без использования скобок (`..`). Тег (*tag*) — часть `foo` перед строковым литералом `'..'` — представляет собой значение функции, которую нам нужно вызвать. На его роль годится любое выражение, дающее в результате функцию, — даже вызов функции, возвращающий другую функцию, например:

```
function bar() {

  return function foo(strings, ...values) {
    console.log( strings );
    console.log( values );
  }
}

var desc = "awesome";
bar()'Everything is ${desc}!';
// [ "Everything is ", "!" ]
// [ "awesome" ]
```

Но что именно передается в функцию `foo(..)`, будучи активированным в качестве тега для строкового литерала?

Первый аргумент — мы дали ему имя `strings` — представляет собой массив из обычных строк (именно они появляются между любыми интерполированными выражениями). В массиве строк мы получаем два значения: `"Everything is "` и `"!"`.

Для удобства в нашем примере все последующие аргументы будут собраны в массив `values` с помощью оператора `...`, который мы рассматривали в разделе «Операторы Spread и Rest» в начале этой главы. Разумеется, их нужно оставить в виде параметров с индивидуальными именами, следующих за параметром `strings`.

Аргументы, собранные в массив `values`, представляют собой результаты уже вычисленных интерполированных выражений, обнаруженных в строковом литерале. Потому очевидно, что в нашем примере массив `values` будет содержать единственный элемент `"awesome"`.

Эти два массива можно представить следующим образом: значения в массиве `values` — разделительные знаки, которые нужно вставить между значениями массива `strings`, после чего мы получим полное интерполированное строковое значение.

Тегированный строковый литерал можно сравнить с этапом обработки, который выполняется после вычисления выражений интерполяции, но до компиляции итогового строкового значения, что дает вам дополнительный контроль над генерацией строки из литерала.

Как правило, тегированная функция строкового литерала (в предыдущем фрагменте это `foo(..)`) должна вычислить соответствующую строку и вернуть ее, чтобы вы смогли использовать тегированный строковый литерал как значение аналогично тому, как происходит в случае с обычными строковыми литералами.

```
function tag(strings, ...values) {
  return strings.reduce( function(s,v,idx){
    return s + (idx > 0 ? values[idx-1] : "") + v;
  }, "" );
}

var desc = "awesome";

var text = tag'Everything is ${desc}!';

console.log( text );           // Everything is awesome!
```

В приведенном фрагменте `tag(..)` — это промежуточная операция, во время которой не совершается никаких специальных преобразований, а всего лишь используется метод `reduce(..)` для циклического просмотра и сращивания/чередования строк и значений тем же способом, что и в случае обычного строкового литерала.

Как же все это применяется на практике? Существует множество нетривиальных вариантов, обсуждение которых выходит за рамки данной книги. В качестве примера мы рассмотрим простое форматирование чисел с помощью значка американского доллара (своего рода примитивную локализацию):

```
function dollabillsyall(strings, ...values) {
  return strings.reduce( function(s,v,idx){
    if (idx > 0) {
      if (typeof values[idx-1] == "number") {
        // смотрите, здесь также используются
        // интерполированные строковые литералы!
        s += '$${values[idx-1].toFixed( 2 )}';
      }
      else {
        s += values[idx-1];
      }
    }

    return s + v;
  }, "" );
}
```

```
}

var    amt1 = 11.99,
      amt2 = amt1 * 1.08,
      name = "Кайл";

var text = dollabillsyall
'Sпасибо за покупку, ${name}! Выбранный вами
Продукт стоил ${amt1}, что вместе с НДС
составляет ${amt2}.'
```

```
console.log( text );
// Спасибо за покупку, Кайл! Выбранный вами
// продукт стоил $11.99, что вместе с НДС
// составляет $12.95.
```

Если значение `number` встречается в массиве `values`, мы располагаем перед ним знак "\$" и форматируем число, оставляя всего два десятичных знака, с помощью метода `toFixed(2)`. В противном случае значение остается необработанным.

Неформатированные строки

В предыдущих фрагментах кода наша тегированная функция получала в качестве первого аргумента массив `strings`. Но этот аргумент содержал небольшой дополнительный фрагмент данных: неформатированные версии всех строк. Доступ к таким значениям дает свойство `.raw`. Рассмотрим пример:

```
function showraw(strings, ...values) {
  console.log( strings );
  console.log( strings.raw );
}

showraw'Hello\nWorld';
// [ "Hello
// World" ]
// [ "HeLlO\nWoRlD" ]
```

В необработанном значении сохранена управляющая последовательность `\n` (`\` и `n` — это отдельные символы), в то время как обработанная версия считает ее одним символом новой строки. К обоим значениям применяется упоминавшаяся выше нормализация концов строк.

В ES6 появилась встроенная функция, которую можно использовать как тег строкового литерала: `String.raw(...)`. Она просто передает неформатированные версии элементов массива `strings`:

```
console.log( 'Hello\nWorld' );  
// Hello  
// World  
  
console.log( String.raw'Hello\nWorld' );  
// Hello\nWorld  
  
String.raw'Hello\nWorld'.length;  
// 12
```

Другие варианты применения тегов строковых литералов связаны с интернационализацией, локализацией и прочими подобными вещами.

Стрелочные функции

Ранее мы уже сталкивались с проблемой, возникающей при связывании функций. Детально она рассматривается в книге *this & Object Prototypes* этой серии. Важно понимать, с какими проблемами сопряжено использование ключевого слова `this` с обычными функциями, так как именно это стало основной причиной появления в ES6 стрелочных функций `=>`.

Для начала давайте посмотрим, чем они отличаются от обычных функций.

```
function foo(x,y) {  
    return x + y;  
}
```

// в сравнении с

```
var foo = (x,y) => x + y;
```

Определение стрелочной функции состоит из списка параметров (их может и не быть, а если их больше одного, они заключаются в скобки (..)), за которым следует маркер => и тело функции.

В последнем примере стрелочная функция выглядит как (x,y) => x + y, а ссылка на нее присвоена переменной foo.

Тело функции заключается в фигурные скобки { .. }, только когда оно содержит более одного выражения или состоит из оператора, не являющегося выражением. Если при единственном выражении вы опускаете фигурные скобки { .. }, предполагается, что перед ним как бы находится оператор return, показанный в приведенном выше фрагменте кода.

Рассмотрим другой пример стрелочной функции:

```
var    f1 = () => 12;  
var    f2 = x => x * 2;  
var    f3 = (x,y) => {  
var    z = x * 2 + y;  
        y++;  
        x *= 3;  
        return (x + y + z) / 2;  
};
```

Стрелочные функции *всегда* представляют собой функциональные выражения; такой вещи, как объявление стрелочной функции, попросту не существует. Кроме того, вы должны понимать, что эти функциональные выражения — анонимные, у них нет именованной ссылки для организации рекурсии или связывания/открепления событий — хотя в разделе «Имена функций» главы 7 описываются правила вывода имен функций в ES6 для отладочных целей.



Стрелочным функциям доступны все возможности обычных функциональных параметров, в том числе значения по умолчанию, де-структуризация, *rest*-параметры и пр.

Стрелочные функции обладают более компактным синтаксисом, что на первый взгляд делает их выгодным вариантом для написания лаконичного кода. Кроме того, создается впечатление, что практически все авторы книг по ES6 (кроме тех, что выходят в нашей серии) немедленно и безоговорочно признали стрелочные функции «новыми функциями».

Примечательно, что практически все примеры стрелочных функций представляют собой короткие, состоящие из одного оператора элементы, скажем, передаваемые в различные места в качестве обратных вызовов. Например:

```
var a = [1,2,3,4,5];  
  
a = a.map( v => v * 2 );  
  
console.log( a );           // [2,4,6,8,10]
```

В случаях со встроенными функциональными выражениями, которые сводятся к работе единственного оператора и возвращению результата, стрелочные функции выглядят как привлекательная и простая альтернатива более многословному синтаксису, сопровождающему ключевое слово *function*.

Многие разработчики приходят в восторг при виде лаконичных примеров, подобных приведенным в этом главе.

Но я хотел бы предостеречь вас от соблазна перейти к стрелочному синтаксису в случае обычных, состоящих из нескольких операторов функций, особенно тех, которые естественным образом выражаются как объявления функций.

Давайте еще раз обратимся к тегированной функции строкового литерала `dollabillsyall(..)`, с которой вы уже работали в преды-

дущем разделе. Попробуем переписать ее с помощью стрелочного синтаксиса.

```
var    dollabillsyall = (strings, ...values) =>
      strings.reduce( (s,v,idx) => {
        if (idx > 0) {
          if (typeof values[idx-1] == "number") {
            // обратите внимание, здесь также используются
            // интерполированные строковые литералы!
            s += `${values[idx-1].toFixed( 2 )}`;
          }
          else {
            s += values[idx-1];
          }
        }
        return s + v;
      }, "" );
```

В данном примере я всего лишь убрал ключевые слова `function`, `return` и некоторые фигурные скобки `{ .. }`, а затем вставил `=>` и ключевое слово `var`. Увеличилась ли после этого читабельность кода? Нет.

На мой взгляд, отсутствие оператора `return` и внешних фигурных скобок `{ .. }` частично маскирует тот факт, что вызов метода `reduce(..)` — единственный оператор в теле функции `dollabillsyall(..)` и что его результат — это предполагаемый результат вызова. Кроме того, тренированный глаз, привыкший для определения границ области видимости искать в коде ключевое слово `function`, теперь должен высматривать маркер `=>`, что куда тяжелее.

Несмотря на то что исключения из этого правила возможны, я бы сказал, что улучшение читабельности после перехода к стрелочному синтаксису обратно пропорционально длине преобразуемой функции. Чем она длиннее, тем меньше пользы от `=>`; чем она короче, тем разумнее использовать `=>`.

Я думаю, что целесообразно прибегать к `=>` в тех местах, где требуются короткие встроенные функциональные выражения, а основ-

ным функциям, имеющим нормальную длину, оставлять привычный вид.

Не только сокращенный синтаксис, но и ключевое слово `this`

Такое внимание к синтаксису => объясняется главным образом возможностью уменьшить количество набираемого текста, убрав из кода ключевые слова `function`, `return` и фигурные скобки `{ .. }`.

Но мы пока не обсудили одну важную деталь. В начале раздела я упомянул, что стрелочные функции => имеют близкое отношение к связыванию с помощью ключевого слова `this`. Более того, они *целенаправленно созданы* для изменения поведения `this` определенным образом, позволяющим решить конкретную и давно наболевшую проблему.

Так что возможность набирать меньше текста — это лишь сопутствующий приятный фактор.

Рассмотрим еще один из ранее встречавшихся примеров:

```
var controller = {
  makeRequest: function(..){
    var self = this;

    btn.addEventListener( "click", function(){
      // ..
      self.makeRequest(..);
    }, false );
  }
};
```

Мы воспользовались приемом `var self = this`, после чего дали ссылке `self.makeRequest(..)`, так как внутри функции обратного вызова, передаваемой в метод `addEventListener(..)`, связывание при помощи ключевого слова `this` будет отличаться от того, что мы имеем в самой функции `makeRequest(..)`. Иначе говоря, из-за ди-

намический природы связывания с помощью ключевого слова `this` мы возвращаемся к предсказуемости лексического контекста через переменную `self`.

Именно в этой ситуации мы наконец видим основное назначение стрелочных функций. Внутри них контекст исполнения, связанный с ключевым словом `this`, становится не динамическим, а лексическим. Если бы в предыдущем фрагменте кода мы воспользовались для обратного вызова стрелочной функцией, ключевое слово `this` предсказуемо дало бы нам требуемый результат.

Рассмотрим пример:

```
var    controller = {
  makeRequest: function(..){
    btn.addEventListener( "click", () => {
      // ..
      this.makeRequest(..);
    }, false );
  }
};
```

Лексически `this` в обратном вызове стрелочной функции из предыдущего фрагмента теперь указывает на то же самое значение, что и в охватывающей функции `makeRequest(..)`. Другими словами, `=>` — это синтаксическая замена объявлению `var self = this`.

В случаях, когда помогает объявление `var self = this` (или функциональный вызов `.bind(this)`), стрелочные функции дают более удобную возможность сделать то же самое. Здорово, не правда ли?

Но все не так просто.

Если `=>` заменяет `var self = this` или `.bind(this)` и это дает нужный результат, догадайтесь, что произойдет, если вы используете `=>` с работающей с `this` функцией, которой *не требуется* объявления `var self = this`? Скорее всего, вы уже поняли, что ничего хорошего в этой ситуации не получится.

Рассмотрим пример:

```
var controller = {  
  makeRequest: (..) => {  
    // ..  
    this.helper(..);  
  },  
  helper: (..) => {  
    // ..  
  }  
};  
  
controller.makeRequest(..);
```

Несмотря на вызов функции в форме `controller.makeRequest(..)`, ссылка `this.helper` не даст результата, так как здесь `this` не указывает на объект `controller`, как должно было бы получиться в обычной ситуации. Куда же нацелен указатель? В данном случае он лексически наследует окружающий контекст. В предыдущем фрагменте имела место глобальная область видимости, поэтому `this` указывал на глобальный объект.

Стрелочные функции обеспечивают лексический контекст не только указателю `this`, но и массиву `arguments` — собственный массив у них отсутствует, вместо него используется унаследованный от предка такой функции, — а также ключевому слову `super` и свойству `new.target` (см. раздел «Классы» в главе 3).

Теперь мы можем сформулировать более конкретные правила применения стрелочных функций.

- При наличии короткого встроеного функционального выражения, состоящего из одного оператора, возвращающего вычисленное значение *и* не имеющего внутри ссылки с ключевым словом `this` или с переменной `self` (рекурсии, связывание/открепление событий), причем только если вы уверены, что эти вещи там никогда не появятся, скорее всего, вы можете переписать код с применением стрелочных функций.

- При наличии внутреннего функционального выражения, связанного с объявлением `var self = this` или с вызовом `.bind(this)` в охватывающей функции, для гарантии корректного связывания с помощью `this` это выражение, скорее всего, можно без проблем превратить в стрелочную функцию.
- При наличии внутреннего функционального выражения, связанного, например, с объявлением `var args = Array.prototype.slice.call(arguments)` в охватывающей функции, для создания лексической копии аргументов это выражение, скорее всего, можно без проблем превратить в стрелочную функцию.
- В остальных случаях — таких как обычные объявления функций, более длинные, состоящие из нескольких операторов функциональные выражения, функции, которым требуется ссылка на лексический именной идентификатор `self` (при рекурсии и т. п.), и все прочие функции, не отвечающие приведенным выше характеристикам, — имеет смысл избегать синтаксиса `c =>`.

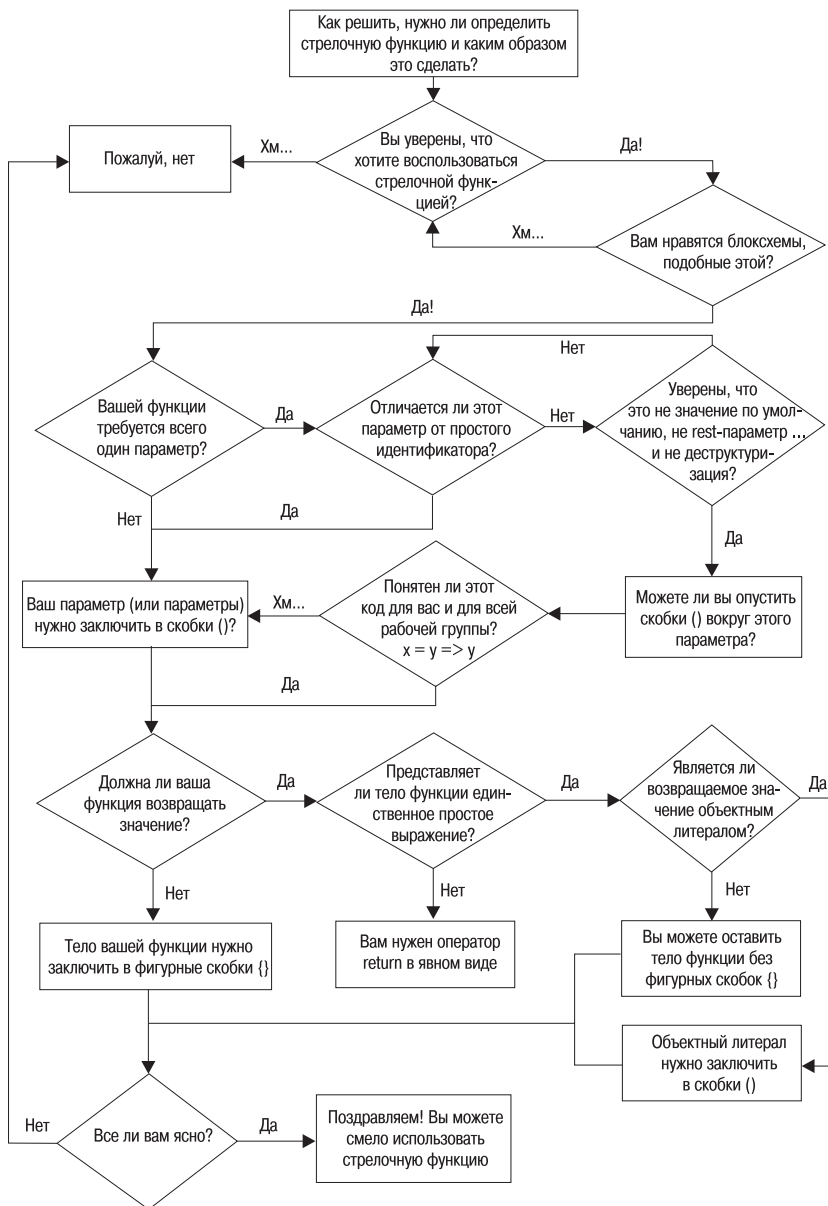
Подытожим: синтаксис `c =>` касается лексического контекста таких вещей, как `this`, `arguments` и `super` при обратных вызовах. Стрелочные функции разработаны специально для решения некоторых часто возникающих проблем.

Не верьте утверждениям, что синтаксис `c =>` главным образом уменьшает количество набираемых символов. Сколько бы кода вы ни набирали, вы в каждый момент должны понимать, зачем нужен тот или иной его фрагмент.



Если у вас есть функция, которую по любой из сформулированных выше причин нельзя переписать с использованием синтаксиса `c =>`, но которая объявлена как часть объектного литерала, — вы можете ее сократить. Об этом будет рассказано в разделе «Краткие методы».

Для тех, кто предпочитает визуальные диаграммы, я проиллюстрирую процесс выбора стрелочных функций.



Цикл `for..of`

Стандарт ES6 содержит не только знакомые всем нам циклы `for` и `for..in`, но и новый цикл `for..of`, работающий с набором значений от *итератора*.

Значение, которое вы просматриваете с помощью цикла `for..of`, должно быть *итерируемым* или же допускающим приведение к итерируемому объекту (см. книгу *Types & Grammar* этой серии). Итерируемым или перебираемым называется объект, способный сформировать итератор, который затем можно использовать в цикле.

Сравним циклы `for..of` и `for..in`, чтобы понять, в чем состоит их отличие:

```
var    a = ["a","b","c","d","e"];

for    (var idx in a) {
    console.log( idx );
}
// 0 1 2 3 4

for    (var val of a) {
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

Как легко заметить, цикл `for..in` перебирает ключи/индексы массива `a`, в то время как цикл `for..of` просматривает его значения.

Вот как выглядел цикл `for..of` из предыдущего фрагмента до появления ES6:

```
var    a = ["a","b","c","d","e"],
        k = Object.keys( a );

for    (var val, i = 0; i < k.length; i++) {
    val = a[ k[i] ];
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

А вот пример цикла из ES6, который, не будучи эквивалентом цикла `for..of`, дает представление о переборе итератора вручную (см. раздел «Итераторы» в главе 3):

```
var    a = ["a", "b", "c", "d", "e"];

for    (var val, ret, it = a[Symbol.iterator]();
      (ret = it.next()) && !ret.done;
) {
    val = ret.value;
    console.log( val );
}
// "a" "b" "c" "d" "e"
```

Механизм работы выглядит следующим образом: цикл `for..of` просит у итерируемого объекта итератор (с помощью встроенного свойства `Symbol.iterator`, см. раздел «Известные символы» в главе 7), а затем снова и снова вызывает этот итератор, присваивая генерируемое им значение итерационной переменной цикла.

Вот стандартные встроенные значения JavaScript, которые по умолчанию являются итерируемыми (или предоставляют такую возможность):

- массивы;
- строки;
- генераторы (см. главу 3);
- коллекции / типизированные массивы (см. главу 5).



С обычными объектами цикл `for..of` не работает, потому что у них отсутствует итератор, причем это сделано целенаправленно. Впрочем, причины принятия того или иного стандарта мы здесь рассматривать не будем. В разделе «Итераторы» главы 3 вы научитесь определять итератор для ваших собственных объектов. Это позволит вам перебирать с помощью цикла `for..of` любые объекты, получая определенный набор значений.

Вот пример циклического перебора символов в обычной строке:

```
for (var c of "hello") {  
    console.log( c );  
}  
// "h" "e" "l" "l" "o"
```

Обычная строка "hello" приведена к контейнерному эквиваленту объекта `String`, который итерируем по умолчанию.

В выражении `for (XYZ of ABC) ..` часть `XYZ` может быть или выражением присваивания, или объявлением, идентичным той же самой части в циклах `for` и `for .. in`. Это позволяет делать вот такие вещи:

```
var    o = {};  
  
for    (o.a of [1,2,3]) {  
    console.log( o.a );  
}  
// 1 2 3  
  
for    ({x: o.a} of [ {x: 1}, {x: 2}, {x: 3} ]) {  
    console.log( o.a );  
}  
// 1 2 3
```

Цикл `for .. of`, как и любой другой цикл, можно остановить, не дожидаясь выполнения всех итераций, с помощью операторов `break`, `continue`, `return` (если он работает внутри функции), а также выбросив исключение. Во всех перечисленных случаях автоматически вызывается функция `return(..)` итератора (если таковая существует), давая итератору возможность выполнить задачи, связанные с возвратом ресурсов, если это необходимо.



Более подробную информацию об итерируемых объектах и итераторах вы найдете в разделе «Итераторы» главы 3.

Регулярные выражения

Посмотрим правде в глаза: регулярные выражения в JS не менялись в течение длительного времени. Так что можно только порадоваться, что в ES6 они получили парочку новых особенностей. Мы кратко рассмотрим их, но эта тема столь обширна, что если вы хотите повторить материал, лучше обратиться к специализированной литературе.

Флаг интерпретации символов Unicode

Стандарт Unicode будет подробно рассматриваться в одноименном разделе данной главы, здесь же мы кратко поговорим о новом флаге `u` для регулярных выражений стандарта ES6+, который включает для них соответствующий символ Unicode.

Строки JavaScript обычно интерпретируются как последовательности 16-битных символов, соответствующих символам из *базовой многоязыковой плоскости* (https://ru.wikipedia.org/wiki/Символы_представленные_в_Юникоде). Но существует множество других символов в кодировке UTF-16, выходящих за границы указанного диапазона, и такие многобайтовые символы могут попадаться в строках.

До появления ES6 сопоставление регулярных выражений осуществлялось исключительно на основе символов из базовой многоязыковой плоскости, потому любой специальный символ рассматривался как два отдельных. Часто это давало далеко не идеальный результат.

Появившийся в ES6 флаг `u` заставляет регулярное выражение обрабатывать строку с интерпретацией символов Unicode (UTF-16) таким образом, что любому специальному символу сопоставляется один элемент.



Вопреки названию, кодировка UTF-16 вовсе не гарантирует, что во всех случаях нам придется иметь дело с 16 битами. В современном Unicode — 21 бит, а в названиях стандартов UTF-8 и UTF-16 содержится лишь приблизительное указание на то, сколько бит будет использоваться в представлении символа.

В качестве примера (непосредственно из спецификации ES6) рассмотрим скрипичный ключ ♩ . Его кодовая точка в Unicode U+1D11E (0x1D11E).

При его появлении в шаблоне регулярного выражения (например, $/\text{♩}/$) стандартная интерпретация с помощью базовой многоязыковой плоскости сопоставит ему два отдельных символа (0xD834 и 0xDD1E). Но новый режим ES6, совместимый с Unicode, предполагает, что запись $/\text{♩}/u$ (или в виде escape-последовательности $/\u{1D11E}/u$) будет совмещать фигурирующий в строке " ♩ " с одним символом.

Возможно, вы не понимаете, почему это важно. В отличной от Unicode базовой многоязыковой плоскости данный шаблон будет рассматриваться как два отдельных символа, но это не мешает обнаружить совпадение в строке, содержащей " ♩ ". Попробуйте и убедитесь сами:

```
 $/\text{♩}/.test( "\text{♩}-clef" ); // true$ 
```

Значение в данном случае имеет длина сопоставляемых символов. Например:

```
 $/^.-clef/.test( "\text{♩}-clef" ); // false$   
 $/^.-clef/u.test( "\text{♩}-clef" ); // true$ 
```

Запись $^.-clef$ в шаблоне означает, что мы ищем всего один символ в начале строки перед обычным текстом "-clef". В стандартном режиме базовой многоязыковой плоскости совпадения не получится (из-за двух символов), а вот при наличии флага u в режиме Unicode оно будет обнаружено.

Также важно отметить, что флаг `u` заставляет такие квантификаторы, как `+` и `*`, влиять на кодовую точку Unicode в целом как на единый символ, а не только как на *более низкий суррогат* (lower surrogate), например на крайнюю правую половину символа.

Аналогичным образом дело обстоит с классами символов, например `/[\p{L}-\p{N}]/u`.



Поведение флага `u` в регулярных выражениях подробно описал Матиас Байненс: <https://mathiasbynens.be/notes/es6-unicode-regex>.

Липкий флаг

Благодаря ES6 у регулярных выражений появился еще один флаг `y`, который часто называют липким режимом. Липкость, по сути, означает, что в начале регулярного выражения находится виртуальный якорь, привязывающий начало поиска к индексу, заданному свойством `lastIndex` регулярного выражения.

Для иллюстрации рассмотрим два регулярных выражения — первое в обычном, а второе в липком режиме:

```
var    re1 = /foo/,
      str = "++foo++";

re1.lastIndex;           // 0
re1.test( str );         // true
re1.lastIndex;           // 0 — не обновлено

re1.lastIndex = 4;
re1.test( str );         // true — проигнорировано 'lastIndex'
re1.lastIndex;           // 4 — не обновлено
```

В данном фрагменте обращают на себя внимание три вещи:

- метод `test(..)` не обращает внимания на значение свойства `lastIndex` и всегда выполняет поиск соответствий с начала введенной строки;

- так как в нашем шаблоне отсутствует якорь начала ввода ^, поиск символов "foo" может выполняться свободным перемещением вперед по всей строке;
- свойство `lastIndex` не обновляется методом `test(..)`.

А вот пример этого же выражения в липком режиме:

```
var    re2 = /foo/y,      // <-- обратите внимание на флаг 'y'
      str = "++foo++";

re2.lastIndex; // 0
re2.test( str );      // false - "foo" не обнаружено
                        // в позиции '0'

re2.lastIndex;      // 0
re2.lastIndex = 2;
re2.test( str );      // true
re2.lastIndex;      // 5 - обновлено после предыдущего
                        // совпадения

re2.test( str );      // false
re2.lastIndex;      // 0 - сброшено после предыдущего
                        // неудачного поиска
```

Вот что мы наблюдаем на этот раз:

- метод `test(..)` использует свойство `lastIndex` как точное и единственное положение в строке `str` для поиска соответствия; никакого перемещения вперед не происходит — у нас или есть соответствие в положении `lastIndex`, или нет;
- обнаружив соответствие, метод `test(..)` обновляет свойство `lastIndex` таким образом, что оно начинает указывать на символ, расположенный сразу после совпавшего. В случае неудачи метод `test(..)` сбрасывает значение свойства `lastIndex` на 0.

Обычные, не липкие шаблоны, которые не привязаны к началу ввода с помощью якоря ^, могут свободно двигаться вперед по введенной строке в поисках соответствия. Липкий же режим огра-

ничивает шаблон поиском совпадения только в позиции, заданной свойством `lastIndex`.

В начале раздела я предложил еще одну трактовку происходящего: флаг `y` помещает в начало шаблона виртуальный якорь, связанный с позицией, которая задана свойством `lastIndex` (иначе говоря, это ограничитель начала поиска).



В предшествующих публикациях по данной теме утверждалось, что подобное поведение аналогично внедрению в шаблон якоря `^` (начало ввода). Но это не совсем точно, о чем я расскажу подробно в разделе «Липкая привязка».

Липкое позиционирование

Может показаться странным ограничением тот факт, что, используя флаг `y` в случае повторяющихся совпадений, вы должны вручную проверять нахождение свойства `lastIndex` в нужном положении, так как возможности просто перемещать строку поиска вперед у вас уже нет.

Вот один из сценариев на этот случай: если вы знаете, что нужное совпадение всегда располагается в позиции, кратной определенному числу (например, 0, 10, 20 и т. д.), вы можете сконструировать шаблон совпадений, а затем вручную устанавливать значение `lastIndex`.

Рассмотрим пример:

```
var    re = /f../y,  
      str = "foo far fad";  
  
str.match( re );           // ["foo"]  
  
re.lastIndex = 10;  
str.match( re );           // ["far"]  
  
re.lastIndex = 20;  
str.match( re );           // ["fad"]
```

Однако при анализе строки, которая не была отформатирована с указанием фиксированных позиций, как в предыдущем случае, невозможно определить, какое значение следует задавать свойству `lastIndex` перед каждым совпадением.

В данном случае есть одна тонкость, связанная с сохранением: для успешного совпадения флаг `y` требует свойства `lastIndex` в конкретной позиции, но вы вовсе не обязаны вручную задавать значение этого свойства.

Вы можете строить выражения таким образом, чтобы в случае основного совпадения они захватывали всё до и после этой точки вплоть до следующей, совпадение с которой вам потребуется.

Так как свойство `lastIndex` будет задано до следующего символа, расположенного за концом совпадающего фрагмента, получится так, что, если вы совместите все до этой точки, `lastIndex` всегда будет в корректной позиции для заданного флагом `y` шаблона, и именно отсюда начнется следующий поиск.



Если вы не в состоянии представить структуру строки ввода в виде некоего шаблона, как показано выше, описанная техника вам вряд ли подойдет, и вы не сможете воспользоваться флагом `y`.

Структурированный ввод строковых данных — наиболее практичный вариант сценария, при котором флаг `y` позволяет реализовать многократный поиск. Рассмотрим пример:

```
var    re = /\d+\.\s(?:.?)?(?:\s|$)/y
      str = "1. foo 2. bar 3. baz";

str.match( re );           // [ "1. foo ", "foo" ]

re.lastIndex;             // 7 — корректное положение!
str.match( re );          // [ "2. bar ", "bar" ]
re.lastIndex;             // 14 — корректное положение!
str.match( re );          // [ "3. baz", "baz" ]
```

Это работает, так как я заранее знал структуру строки ввода: перед совпадением с заданными символами ("foo" и т. п.) всегда присутствует численный префикс, например "1. ", а после них располагается или пробел, или конец строки (якорь \$). Соответственно, сконструированное мной регулярное выражение фиксирует все вышеперечисленное при каждом основном поиске, а затем я использую совпадающую группу (), чтобы разделить нужные мне данные.

После первого совпадения ("1. foo ") свойство `lastIndex` получает значение 7, что уже является позицией, необходимой для поиска следующего соответствия "2. bar " и т. д.

Если вы собираетесь пользоваться флагом `u` для поиска повторяющихся совпадений, скорее всего, вам придется задуматься об автоматическом позиционировании свойства `lastIndex`, как это было сделано выше.

Сравнение липкого и глобального поиска

Некоторые читатели, вероятно, знают, что существует возможность симитировать поиск, связанный со свойством `lastIndex`, с помощью задающего глобальный поиск флага `g` и метода `exec()`, например:

```
var    re = /o+./g,           // <-- обратите внимание на флаг 'g'!
      str = "foot book more";

re.exec( str );               // ["oot"]
re.lastIndex;                 // 4

re.exec( str );               // ["ook"]
re.lastIndex;                 // 9

re.exec( str );               // ["or"]
re.lastIndex;                 // 13

re.exec( str );               // null — больше совпадений нет!
re.lastIndex;                 // 0 — теперь начинаем сначала!
```

Хотя шаблон, заданный при помощи флага `g`, ищет соответствия посредством метода `exec(..)` с текущего значения свойства `lastIndex` и обновляет его после каждого совпадения (или отсутствия результата), это не то же самое поведение, которое дает флаг `y`.

Обратите внимание, что расположенный в позиции 6 фрагмент `"ook"` был обнаружен во время второго вызова метода `exec(..)`, в то время как свойство `lastIndex` имело значение 4 (с конца предыдущего совпадения). Почему так получилось?

Как уже было сказано выше, обычный поиск осуществляется путем смещения вперед по строке. Выражение в липком режиме в данном случае не дало бы результата, так как для него смещение невозможно.

Кроме нежелательного в ряде случаев поведения, связанного со смещением вперед по строке, использование флага `g` вместо `y` имеет еще один недостаток. Дело в том, что флаг `g` меняет поведение некоторых методов поиска, например `str.match(re)`.

Рассмотрим пример:

```
var    re = /o+./g,           // <-- обратите внимание на флаг
'g'!
    str = "foot book more";

str.match( re );              // ["oot", "ook", "or"]
```

Видите, каким образом все совпадения возвращаются одновременно? Иногда это нормально, но бывают и случаи, когда такое поведение не требуется.

Липкий флаг `y` последовательно выдает вам результаты поиска с помощью таких методов, как `test(..)` и `match(..)`. Нужно только убедиться, что свойство `lastIndex` всегда находится в корректной позиции.

Липкая привязка

Как уже было написано, липкий режим не следует рассматривать как внедрение в шаблон якоря `^`. В регулярных выражениях этот якорь имеет совершенно определенный смысл, который *не меняется* после входа в липкий режим. Он *всегда* указывает на начало строки ввода и никак не связан со свойством `lastIndex`.

Кроме недостаточно точной документации по данной теме, укреплению заблуждений способствует тот факт, что до появления ES6 в Firefox в ходе экспериментов с липким режимом якорь `^` оказался на много лет *связан* со свойством `lastIndex`.

В ES6 ситуацию решили поменять. Теперь якорь `^` в шаблоне означает исключительно начало ввода.

В результате такие шаблоны, как `/^foo/y`, будут находить соответствие "foo" только в начале строки при условии *разрешенного поиска в этом месте*. Если `lastIndex` не равняется 0, поиск не даст результатов. Рассмотрим пример:

```
var    re = /^foo/y,
      str = "foo";

re.test( str );           // true
re.test( str );           // false
re.lastIndex;             // 0 — сброс после неудачи

re.lastIndex = 1;
re.test( str );           // false — сбой позиционирования
re.lastIndex;             // 0 — сброс после неудачи
```

Подведем итог: флаг `y` плюс якорь `^` плюс `lastIndex > 0` — несовместимая комбинация, при которой поиск всегда будет безрезультатным.



Если флаг `у` никак не влияет на `^`, то флаг `т`, активирующий многострочный режим, меняет поведение якоря. При наличии этого флага `^` означает начало ввода или начало текста после перехода на новую строку. Так что, соединив в одном шаблоне флаги `у` и `т`, вы сможете обнаружить в строке несколько совпадений, связанных с якорем `^`. Однако тут важно помнить одну вещь. Так как флаг `у` активирует липкий режим, необходимо следить за тем, чтобы свойство `lastIndex` каждый раз указывало на корректную позицию в новой строке (вероятно, это лучше делать путем сопоставления с концом строки), иначе обнаружить все последующие совпадения попросту не получится.

Флаги регулярных выражений

До появления ES6, если вы хотели посмотреть, какие флаги добавлены к объекту регулярного выражения, вам приходилось извлекать их путем анализа — вероятно, с помощью другого регулярного выражения — из свойства `source`. Например:

```
var re = /foo/ig;

re.toString();           // "/foo/ig"

var flags = re.toString().match( /\s*([gim]*)$/ )[1];

flags;                   // "ig"
```

В ES6 эти значения можно получить напрямую благодаря новому свойству `flags`:

```
var re = /foo/ig;

re.flags;                // "gi"
```

Есть небольшой нюанс. Спецификация ES6 требует, чтобы флаги выражения перечислялись в порядке “`gimuy`”, независимо от их последовательности в исходном шаблоне. Вот в чем причина разницы между `/ig` и “`gi`”.

При этом порядок перечисленных флагов не имеет значения.

Кроме того, в ES6 конструктор `RegExp(..)` стал совместим с флагами в случаях, когда ему передают существующее регулярное выражение:

```
var re1 = /foo*/y;
re1.source;           // "foo*"
re1.flags;            // "y"

var re2 = new RegExp( re1 );
re2.source;           // "foo*"
re2.flags;            // "y"

var re3 = new RegExp( re1, "ig" );
re3.source;           // "foo*"
re3.flags;            // "gi"
```

До появления ES6 конструкция `re3` приводила к появлению ошибки, теперь же можно просто переопределить флаги при дублировании.

Расширения числовых литералов

До ES5 числовые литералы выглядели следующим образом: официально спецификация восьмеричной формы отсутствовала, она допускалась только как расширение, относительно которого браузеры фактически пришли к соглашению:

```
var    dec = 42,
      oct = 052,
      hex = 0x2a;
```



Хотя вы указываете число с различными основаниями, его математическое значение, которое сохраняется в переменной, а также интерпретация по умолчанию всегда имеют основание 10. Так, во всех трех переменных из предыдущего фрагмента сохранено значение 42.

Чтобы показать, что 052 — это нестандартное расширение формы, рассмотрим следующий пример:

```
Number( "42" );           // 42
Number( "052" );          // 52
Number( "0x2a" );         // 42
```

Стандарт ES5 допускал расширенную для браузеров восьмеричную форму (в том числе и подобные разночтения). Запрещалась восьмеричная форма литерала (052) только в строгом режиме. Такое ограничение было введено в основном потому, что разработчики по привычке (оставшейся со времен программирования на других языках) нередко предворяли числа с основанием 10 символом `0` с целью выравнивания кода, тем самым полностью меняя их значения.

В ES6 работа над представлением числовых литералов с основанием, отличающимся от 10, была продолжена. Теперь существует официальная восьмеричная форма, скорректированная шестнадцатеричная форма и совершенно новая бинарная форма. Для обеспечения совместимости старая восьмеричная форма (052) будет допускаться в нестрогом режиме; хотя она и отсутствует в спецификации, но лучше ее теперь не пользоваться.

Вот новые формы числовых литералов из ES6:

```
var    dec = 42,
      oct = 0o52,           // или '0052' :(
      hex = 0x2a,          // или '0X2a' :/
      bin = 0b101010;      // или '0B101010' :/
```

Единственная допустимая десятичная форма имеет основание 10. Восьмеричные, шестнадцатеричные и бинарные представления допустимы только для целых чисел.

Все строковые представления данных форм можно привести к числовому эквиваленту:

```
Number( "42" );           // 42
Number( "0o52" );         // 42
Number( "0x2a" );         // 42
Number( "0b101010" );     // 42
```

Есть еще одна возможность, которая, строго говоря, появилась до ES6, но по большому счету известна не широко. Дело в том, что подобное преобразование выполняется и в обратную сторону (по крайней мере, в некоторых случаях):


```
var    a = 42;

a.toString();              // "42" – кроме того, 'a.toString( 10 )'
a.toString( 8 );          // "52"
a.toString( 16 );         // "2a"
a.toString( 2 );          // "101010"
```

Представить число в такой форме можно с любым основанием от 2 до 36, хотя ситуации, когда нужны основания, отличные от стандартных 2, 8, 10 и 16, встречаются крайне редко.

Unicode

Сразу скажу, что этот раздел ни в коем случае нельзя считать исчерпывающим ресурсом по Unicode. Здесь мы поговорим только о том, что вам требуется знать о связанных с Unicode *изменениях* в стандарте. Много и подробно про JS и Unicode писал Матиас Байненс (см. <https://mathiasbynens.be/notes/javascript-unicode> и <http://fluentconf.com/javascript-html-2015/public/content/2015/02/18-javascript-lovesunicode>).

Диапазон от 0x0000 до 0xFFFF содержит все стандартные печатные символы (из разных языков), которые вам только доводилось видеть или использовать. Эта группа символов называется *базовой многоязыковой плоскостью* (BMP — basic multilingual plane). Она содержит даже такие забавные символы, как, например, снеговик:  (U+2603).

Существует множество расширенных символов Unicode, расположенных за пределами BMP, диапазон которых простирается до 0x10FFFF. Их часто называют *астральными*. Это имя было дано 16 плоскостям, включающим в себя символы, не входящие в BMP. Выше вы уже видели примеры астральных символов — ♪ U+1D11E и ☺ U+1F4A9.

До появления ES6 строки JavaScript определяли символы Unicode через escape-последовательности. Например:

```
var snowman = "\u2603";  
console.log( snowman ); // "☺"
```

Но escape-последовательность \uxxxx в Unicode поддерживает только четыре шестнадцатеричных символа, поэтому представить подобным способом можно только содержимое BMP. Для представления с помощью escape-последовательности астральных символов Unicode раньше приходилось прибегать к *суррогатной паре* (surrogate pair), которая, по сути, была двумя записанными рядом кодовыми значениями, интерпретируемыми JS как один астральный символ:

```
var gclef = "\uD834\uDD1E";  
console.log( gclef ); // "🎸"
```

Стандарт ES6 дал нам новую форму escape-последовательности Unicode (в строках и регулярных выражениях), так называемую *escape-последовательность кодовых точек*:

```
var gclef = "\u{1D11E}";  
console.log( gclef ); // "🎸"
```

Легко заметить, что разница заключается в наличии фигурных скобок { }, и это позволяет использовать произвольное число шестнадцатеричных символов. Для представления максимально возможного значения кодовой точки в Unicode (0x10FFFF) нужно шесть цифр.

Операции со строками, поддерживающие Unicode

По умолчанию работающие со строками операции и методы JavaScript не чувствительны к астральным символам в строковых значениях. Поэтому каждый ВМР-символ они трактуют индивидуально, даже две части суррогатной пары, составляющие один астральный символ. Рассмотрим пример:

```
var snowman = "☹";  
snowman.length;           // 1  
  
var gclef = "♫";  
gclef.length;             // 2
```

Как точно вычислить длину такой строки? В данном сценарии хорошо работает следующий трюк:

```
var gclef = "♫";  
  
[...gclef].length;        // 1  
Array.from( gclef ).length; // 1
```

Ранее в разделе «Цикл `for..of`» упоминалось, что строки в ES6 обладают встроенным итератором. Он умеет работать с Unicode и автоматически выводит астральный символ как одно значение. Мы воспользуемся этим и применим к литералу массива оператор `...`, который создает массив символов строки. Затем нам останется только определить длину полученного массива. Появившийся в ES6 метод `Array.from(.)`, по сути, делает то же самое, что и `[...XYZ]`, но его мы будем детально рассматривать в главе 6.



Следует отметить, что конструирование и применение итератора только для получения длины строки — неоправданно ресурсоемкая операция, особенно в сравнении с теоретически оптимизированной под эту задачу утилитой или свойством.

К сожалению, в целом ситуацию нельзя назвать простой и однозначной. В дополнение к суррогатным парам (о которых способен позаботиться строковый итератор) существуют специальные кодовые точки Unicode, и их особое поведение учесть куда сложнее. К примеру, есть некоторое число кодовых точек, модифицирующих предыдущий символ. Это так называемые *комбинированные диакритические знаки* (Combining Diacritical Marks).

Рассмотрим два варианта строкового вывода:

```
console.log( s1 );           // "é"
console.log( s2 );           // "é"
```

Символы выглядят одинаково, но они разные! Вот как были созданы `s1` и `s2`:

```
var    s1 = "\xE9",
      s2 = "e\u0301";
```

Как вы, наверное, догадались, наш предыдущий трюк с определением длины в случае `s2` не работает:

```
[...s1].length;             // 1
[...s2].length;             // 2
```

Что в такой ситуации можно сделать? Перед тем как поинтересоваться длиной, мы выполним *нормализацию значения Unicode* с помощью появившегося в ES6 метода `String#normalize(..)` (который будет подробно рассмотрен в главе 6):

```
var    s1 = "\xE9",
      s2 = "e\u0301";

s1.normalize().length;      // 1
s2.normalize().length;      // 1

s1 === s2;                  // false
s1 === s2.normalize();      // true
```


По сути, метод `normalize(...)` берет последовательность, например `"e\u0301"`, и нормализует ее до `"\xE9"`. Нормализация позволяет даже объединять несколько соседних комбинируемых знаков, если существует подходящий символ Unicode:

```
var    s1 = "o\u0302\u0300",
        s2 = s1.normalize(),
        s3 = "ö";

s1.length;           // 3
s2.length;           // 1
s3.length;           // 1

s2 === s3;           // true
```

К сожалению, идеальный результат в данном случае недостижим даже с помощью нормализации. При наличии нескольких комбинируемых знаков, модифицирующих один символ, результат определения длины способен оказаться совсем не таким, какой вы ожидаете, потому что нормализованного символа, представляющего собой комбинацию всех знаков, может попросту не существовать. Например:

```
var s1 = "e\u0301\u0330";

console.log( s1 );           // "é"
s1.normalize().length;       // 2
```

Чем глубже мы копаем, тем очевиднее становится, что дать точное определение понятию «длина» крайне сложно. То, что мы видим как один символ — более точно это называется *графемой*, — не всегда является таковым с точки зрения программной обработки.



Если вы хотите посмотреть, насколько глубока эта кроличья нора, почитайте про алгоритм, позволяющий определить границу кластера графемы.

Позиционирование символов

Сложности бывают не только с определением длины. К примеру, что конкретно означает фраза «символ, который располагается в позиции 2»? До появления ES6 мы получали наивный ответ при помощи метода `charAt(..)`, игнорировавшего неделимость астральных символов и не принимавшего во внимание комбинируемые знаки.

Рассмотрим пример:

```
var    s1 = "abc\u0301d",
        s2 = "ab\u0107d",
        s3 = "ab\u{1d49e}d";

console.log( s1 );           // "abċd"
console.log( s2 );           // "abċd"
console.log( s3 );           // "abĸd"

s1.charAt( 2 );               // "ċ"
s2.charAt( 2 );               // "ċ"
s3.charAt( 2 );               // "" <-- непечатаемый суррогат
s3.charAt( 3 );               // "" <-- непечатаемый суррогат
```

Появилась ли в ES6 умеющая работать с Unicode версия метода `charAt(..)`? К сожалению, нет. На момент написания книги такой метод только предложили — он будет рассматриваться при подготовке следующей версии стандарта.

Впрочем, то, что мы узнали в предыдущем разделе (разумеется, с учетом всех перечисленных там ограничений), позволит нам получить нужный ответ средствами ES6:

```
var    s1 = "abc\u0301d",
        s2 = "ab\u0107d",
        s3 = „ab\u{1d49e}d";

[...s1.normalize()][2];      // "ċ"
[...s2.normalize()][2];      // "ċ"
[...s3.normalize()][2];      // "ĸ"
```



Еще раз напомним момент, о котором уже предупреждал вас выше: конструирование и применение итератора каждый раз, когда вам требуется получить информацию об одном символе, — не самое разумное действие с точки зрения производительности. Будем надеяться, что в следующих версиях стандарта появятся встроенные и оптимизированные под выполнение этой задачи функциональные особенности.

А как насчет умеющей работать с Unicode версии метода `charAt(..)`? В ES6 появился метод `codePointAt(..)`:

```
var    s1 = "abc\u0301d",
        s2 = "ab\u0107d",
        s3 = "ab\u{1d49e}d";

s1.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s2.normalize().codePointAt( 2 ).toString( 16 );
// "107"

s3.normalize().codePointAt( 2 ).toString( 16 );
// "1d49e"
```

Возможно ли преобразование в обратном направлении? Умеющая работать с Unicode версия метода `String.fromCharCode(..)` в ES6 называется `String.fromCodePoint(..)`:

```
String.fromCodePoint( 0x107 );           // "ć"

String.fromCodePoint( 0x1d49e );         // "ġ"
```

Подождите, мы же просто можем скомбинировать метод `String.fromCodePoint(..)` и код, полученный методом `PointAt(..)`, получив таким образом распознающую Unicode версию вышеупомянутого `charAt(..)`!

```
var    s1 = "abc\u0301d",
        s2 = "ab\u0107d",
        s3 = "ab\u{1d49e}d";
```

```
String.fromCodePoint( s1.normalize().codePointAt( 2 ) );  
// "ć"
```

```
String.fromCodePoint( s2.normalize().codePointAt( 2 ) );  
// "ć"
```

```
String.fromCodePoint( s3.normalize().codePointAt( 2 ) );  
// "ċ"
```

Существуют и другие строковые методы, которых мы не касались. Это, в частности, `toUpperCase()`, `toLowerCase()`, `substring(..)`, `indexOf(..)`, `slice(..)` и пр. Ни один из них не был подвергнут изменениям для обеспечения полноценной работы с Unicode, поэтому со строками, содержащими астральные символы, следует совершать действия крайне осторожно — возможно, вообще лучше избегать перечисленных методов!

Кроме того, есть несколько строковых методов, которые используют регулярные выражения. В частности, это `replace(..)` и `match(..)`. К счастью, стандарт ES6 позволяет распознавать такие символы в регулярных выражениях. Данная возможность была рассмотрена в разделе «Флаг для интерпретации символов Unicode».

Вот и все, что вам нужно знать. Поддержка строк Unicode в JavaScript после появления ES6 стала значительно лучше (хотя до идеала ей еще далеко) благодаря дополнениям, рассмотренным выше.

Unicode в именах идентификаторов

Символы Unicode можно использовать в именах идентификаторов (переменных, свойств и т. п.). До появления ES6 это осуществлялось с помощью escape-последовательностей Unicode, например:

```
var \u03A9 = 42;
```

```
// то же, что и var Ω = 42;
```

В ES6, кроме этого, можно использовать ранее рассмотренный синтаксис escape-последовательностей кодовых точек:

```
var \u{2B400} = 42;  
  
// то же, что и var 𐝀 = 42;
```

То, какие символы Unicode можно использовать в именах идентификаторов, определяется сложным набором правил. Более того, некоторые символы допустимы только при условии, что они не стоят первыми в именах идентификаторов.



Эту тему подробно осветил Матиас Байненс (см. <https://mathiasbynens.be/notes/javascript-identifiers-es6>).

Причины использования столь необычных символов, как правило, оказываются нетривиальными. Обычно все-таки лучше избегать кода, опирающегося на малораспространенные функциональные особенности.

Тип данных Symbol

Благодаря ES6 в JavaScript впервые за долгое время появился новый примитивный тип данных: `symbol`. В отличие от остальных примитивных типов он не может быть представлен в форме литерала.

Вот как он создается:

```
var sym = Symbol( "необязательное описание" );  
  
typeof sym;           // "symbol"
```

Следует обратить внимание на три момента.

- Использовать оператор `new` с функцией `Symbol(..)` нельзя. Это не конструктор, и вы создаете не объект.
- Передавать функции `Symbol(..)` параметр необязательно. Передаваемый параметр представляет собой строку с описанием назначения символа.
- Оператор `typeof` выводит новое значение ("`symbol`"), которое является основным способом идентификации символа.

Описание, если оно дается, используется исключительно для перевода представления символа в строку:

```
sym.toString();           // "Symbol(какое-то необязательное  
                           описание)"
```

Аналогично тому, как примитивные строковые значения не являются экземплярами класса `String`, символы — не экземпляры класса `Symbol`. Если по какой-то причине вы хотите сконструировать для символьного значения объект-обертку, можно сделать следующее:

```
sym instanceof Symbol;           // false  
  
var symObj = Object( sym );  
symObj instanceof Symbol;        // true  
  
symObj.valueOf() === sym;        // true
```



В приведенном фрагменте переменные `symObj` и `sym` взаимозаменяемы; во всех местах, где фигурируют символы, может использоваться любая из этих форм. Но особых причин предпочесть объект-обертку (`symObj`) примитиву (`sym`) нет. Как и в случае с другими примитивами, лучше выбрать вариант `sym`, а не `symObj`.

Внутреннее значение самого символа — которое еще называют его именем — скрыто из кода, и получить его нельзя. Его можно представить как автоматически генерируемое, уникальное (внутри вашего приложения) строковое значение.

Но если значение скрыто и получить его нельзя, зачем вообще нужен тип `symbol`?

Главным образом он применяется для создания напоминающего строку значения, которое не конфликтует ни с каким другим. Его можно использовать, к примеру, как константу, представляющую имя события:

```
const EVT_LOGIN = Symbol( "event.login" );
```

После этого `EVT_LOGIN` подставляется вместо строкового литерала `"event.login"`:

```
evthub.listen( EVT_LOGIN, function(data){  
    // ..  
} );
```

Удобство в данном случае состоит в том, что символ `EVT_LOGIN` имеет значение, которое не может быть продублировано (случайно или намеренно) ни одним другим значением, то есть перепутать, какое именно событие инициируется или обрабатывается, попросту нельзя.



Сервис `evthub` в последнем примере предполагает, что символьное значение из аргумента `EVT_LOGIN` будет напрямую использоваться как свойство/ключ в каком-то внутреннем объекте (хеше), который отслеживает обработчики событий. Если вместо этого сервису `evthub` потребовалось бы воспользоваться символьным значением как реальной строкой, первым делом нужно было бы выполнить явное приведение методом `String(..)` или `toString()`, потому что неявное приведение символов к строкам невозможно.

Символ допускается использовать в объектах в качестве имени свойства/ключа, например особого свойства, которое будет интерпретироваться как скрытое или как метасвойство. Но важно понимать, что, вопреки вашим намерениям, *на самом деле* оно не будет ни скрытым, ни неизменным.

Рассмотрим модуль, реализующий поведение шаблона-одиночки (singleton), то есть такого, который можно создать всего один раз:

```
const INSTANCE = Symbol( "instance" );

function HappyFace() {
  if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

  function smile() { .. }

  return HappyFace[INSTANCE] = {
    smile: smile
  };
}

var    me = HappyFace(),
      you = HappyFace();

me === you;           // true
```

Символьное значение `INSTANCE` в данном случае представляет собой почти скрытое свойство, напоминающее метасвойство, которое статически хранится в объекте-функции `HappyFace()`.

В качестве альтернативы можно было воспользоваться обычным свойством `__instance` и получить аналогичное поведение. Применение символа попросту улучшает стиль метапрограммирования, поскольку позволяет хранить свойство `INSTANCE` отдельно от остальных, обычных свойств.

Реестр символов

Использование символов имеет свои особенности. Скажем, в последних фрагментах кода переменные `EVT_LOGIN` и `INSTANCE` пришлось сохранить во внешней области видимости (и даже в глобальной), потому что они должны были находиться в открытом доступе для всех частей кода, которым они могли потребоваться.

Чтобы дать коду доступ к символам, их значения следует создать в *глобальном реестре* (global symbol registry). Например:

```
const EVT_LOGIN = Symbol.for( "event.login" );

console.log( EVT_LOGIN );           // Symbol(event.Login)
```

И еще:

```
function HappyFace() {
    const INSTANCE = Symbol.for( "instance" );

    if (HappyFace[INSTANCE]) return HappyFace[INSTANCE];

    // ..

    return HappyFace[INSTANCE] = { .. };
}
```

Метод `Symbol.for(...)` проверяет в глобальном реестре, хранится ли там символ с заданным описанием, и, обнаружив, возвращает его, а в противном случае — создает. Другими словами, глобальный реестр интерпретирует значения символов по тексту описания.

Одновременно это означает, что любая часть приложения может затребовать символ из реестра с помощью метода `Symbol.for(...)` при условии, что имя описания совпадает.

По иронии судьбы, символы в основном предназначались для замены в приложениях *магических строк* (случайных строковых значений, наделенных особым смыслом). Однако вы исправно используете *магические* строковые значения для описаний, чтобы идентифицировать/локализовать эти описания в глобальном реестре символов!

Чтобы избежать случайных пересечений, вы, скорее всего, будете стараться сделать описания ваших символов уникальными. Проще всего этого можно достичь, добавив к описанию префикс/контекст/информацию о пространстве имен.

К примеру, рассмотрим следующую функцию:

```
function extractValues(str) {
    var    key = Symbol.for( "extractValues.parse" ),
          re = extractValues[key] ||
                /[^\s]+?=[^\s]+?(?=[\s]|$)/g,
          values = [], match;

    while (match = re.exec( str )) {
        values.push( match[1] );
    }

    return values;
}
```

В данном случае мы используем магическое строковое значение "extractValues.parse", так как оно достаточно уникально и вряд ли у какого-нибудь другого символа из реестра будет такое же описание.

Если кто-нибудь из пользователей захочет переопределить выполняющее анализ регулярное выражение, он тоже сможет воспользоваться реестром символов:

```
extractValues[Symbol.for( "extractValues.parse" )] =
    /..какой-то шаблон../g;

extractValues( "..какая-то строка.." );
```

Реестр не только служит вспомогательным средством, но и на глобальном уровне сохраняет значения символов. Все, что мы выше видели, можно было сделать, используя в качестве ключа магическую строку "extractValues.parse", а не сам символ. В данном случае улучшения происходят не на функциональном уровне, а на уровне метапрограммирования.

Возможно, вам когда-нибудь придется использовать сохраненное в реестре значение символа, чтобы узнать текст его описания (ключ). Такая необходимость возникает, например, когда требует-

ся объяснить другой части приложения способ нахождения символа в реестре, потому что значение этого символа передать невозможно.

Восстановить текст описания (ключ) зарегистрированного символа позволяет метод `Symbol.keyFor(..)`:

```
var s = Symbol.for( "something cool" );

var desc = Symbol.keyFor( s );
console.log( desc );           // "something cool"

// снова получим символ из реестра
var s2 = Symbol.for( desc );

s2 === s;                     // true
```

Символы как свойства объектов

Символ, который используется как свойство/ключ объекта, сохраняется особым образом и не отображается при обычном перечислении свойств:

```
var o = {
  foo: 42,
  [ Symbol( "bar" ) ]: "hello world",
  baz: true
};

Object.getOwnPropertyNames( o ); // [ "foo", "baz" ]
```

Вот способ, позволяющий получить символьное свойство объекта:

```
Object.getOwnPropertySymbols( o ); // [ Symbol(bar) ]
```

Очевидно, что символьное свойство на самом деле не является скрытым или недоступным, так как его всегда можно увидеть с помощью метода `Object.getOwnPropertySymbols(..)`.

Встроенные символы

В ES6 существует ряд предопределенных, встроенных символов, обеспечивающих различные поведения значениям объектов JavaScript. Но эти символы *отсутствуют* в глобальном реестре.

Они хранятся как свойства объекта-функции `Symbol`. Например, в разделе «Цикл `for..of`» этой главы вы познакомились со значением `Symbol.iterator`:

```
var a = [1,2,3];  
  
a[Symbol.iterator]; // native function
```

Для ссылки на эти встроенные символы в спецификации используется префикс `@@`. Вот наиболее распространенные варианты: `@@iterator`, `@@toStringTag`, `@@toPrimitive`. В стандарте определены и другие встроенные символы, но они используются довольно редко.



Подробную информацию об этих символах вы найдете в главе 7, где рассказывается, каким образом они применяются в метапрограммировании.

Подводим итоги

Стандарт ES6 добавил в JavaScript множество новых синтаксических форм, так что изучать вам придется изрядный объем материала.

Большинство из них спроектированы, чтобы облегчить проблемы распространенных идиом программирования, к примеру таких, как присвоение значений по умолчанию функциональным параметрам или компоновка «оставшихся» параметров в массив. Мощным инструментом, позволяющим точнее выполнять присваивание

значений из массивов и вложенных объектов, стала деструктуризация.

Такие функциональные особенности, как стрелочные функции `=>`, на первый взгляд кажутся попытками обеспечить лаконичный и элегантный синтаксис, но на самом деле обладают крайне специфическим поведением и должны использоваться только в определенных ситуациях.

Завершает синтаксическую эволюцию ES6 расширенная поддержка стандарта Unicode, дополнительные возможности работы с регулярными выражениями и новый примитивный тип `symbol`.

3 Структура

Одно дело — написать код JS, а совсем другое — структурировать его. Распространенные шаблоны структуризации и повторного использования играют важную роль в улучшении читабельности и понятности кода. Помните, что код в такой же степени ориентирован на передачу информации другим разработчикам, как и на предоставление инструкций компьютеру.

В ES6 появилось несколько важных функциональных особенностей, позволивших значительно усовершенствовать эти шаблоны. К ним относятся итераторы, генераторы, модули и классы.

Итераторы

Итератором (iterator) называется структурированный шаблон, предназначенный для последовательного извлечения информации из источника. В программировании подобные шаблоны используются давно. Разумеется, и на JS с незапамятных времен проектируются и реализуются итераторы, так что это далеко не новая тема.

В ES6 для итераторов появился неявный стандартизованный интерфейс. Многие встроенные структуры JavaScript предоставляют

итератор, реализующий этот стандарт. Вы можете сконструировать и собственные версии итераторов, если будете придерживаться стандарта для обеспечения максимальной совместимости.

Итераторы позволяют упорядоченно, последовательно извлекать данные.

Например, можно реализовать сервис, который при каждом запросе будет генерировать новый уникальный идентификатор; или формировать бесконечный ряд значений, прокручиваемых в фиксированном списке циклическим способом; или присоединить итератор к результатам запроса к базе данных, чтобы по одной извлекать новые строки.

Есть еще одна особенность, которая, как правило, не используется в JS. На итераторы можно посмотреть как на контролирующее поведение, действующее пошагово. Иллюстрацию этого вы найдете в разделе «Генераторы» данной главы, хотя подобное реализуемо и без генераторов.

Интерфейсы

На момент написания книги раздел 25.1.1.2 спецификации ES6 описывал интерфейс `Iterator` как отвечающий следующим требованиям:

```
Iterator [обязательные параметры]  
  next() {метод}: загружает следующий IteratorResult
```

Два дополнительных параметра для расширения некоторых итераторов:

```
Iterator [необязательные параметры]  
  return() {метод}: останавливает итератор и возвращает  
    IteratorResult  
  throw() {метод}: сообщает об ошибке и возвращает IteratorResult
```

Интерфейс `IteratorResult` определен следующим образом:

IteratorResult

`value {свойство}`: значение на текущей итерации или окончательное возвращаемое значение (не обязательно, если это `'undefined'`)
`done {свойство}`: тип `boolean`, показывает состояние выполнения



Я называю эти интерфейсы неявными не потому, что они в явном виде не указаны в спецификации — они указаны! — просто они не появляются в качестве доступных коду непосредственных объектов. В ES6 язык JavaScript не поддерживает понятие «интерфейсов», поэтому их появление в вашем собственном коде является откровенно условным. Тем не менее в том месте, где ожидается итератор — например, в цикле `for...of`, — ваш код должен выглядеть в соответствии с этими интерфейсами, или он не будет работать.

Существует также интерфейс `Iterable`, описывающий объект, который должен уметь генерировать итераторы:

Iterable

`@@iterator()` {метод}: генерирует `Iterator`

В разделе «Встроенные символы» главы 2 упоминался `@@iterator` — специальный встроенный символ, представляющий метод, который умеет генерировать итераторы для объекта.

IteratorResult

Интерфейс `IteratorResult` определяет, что возвращаемое значение любой из операций итератора будет представлять собой объект следующего вида:

```
{ value: .. , done: true / false }
```

Встроенные итераторы всегда возвращают значения в таком виде, но если нужно, ничто не мешает добавить и другие свойства.

Например, пользовательский итератор может вставлять в результирующий объект дополнительные метаданные (например, откуда

взялись данные, сколько времени заняло их извлечение, каково время жизни кэша и частота для следующего подходящего запроса и т. п.).



С технической точки зрения значение считается необязательным, если в иной форме оно рассматривается как отсутствующее или незаданное, например как в случае значения `undefined`. Так как доступ к `res.value` даст нам `undefined` вне зависимости от того, есть там значение или нет, наличие/отсутствие данного свойства относится, скорее, к деталям реализации или оптимизации, а не к функциональным вопросам.

Метод `next()`

Рассмотрим итерируемый массив и итератор, который он создает для работы со своими значениями:

```
var arr = [1,2,3];

var it = arr[Symbol.iterator]();

it.next();           // { value: 1, done: false }
it.next();           // { value: 2, done: false }
it.next();           // { value: 3, done: false }

it.next();           // { value: undefined, done: true }
```

Каждый раз, когда связанный со свойством `Symbol.iterator` (оно рассматривается в главах 2 и 7) метод вызывается со значением `arr`, он создает свежий итератор. Большинство структур будут давать аналогичный результат, в том числе встроенные в JS структуры данных.

Но структура, которая, к примеру, является получателем очереди событий, может создать лишь единственный итератор (шаблон-одиночка). Еще бывают структуры, допускающие в каждый момент времени только один итератор. В этом случае перед созданием нового итератора следует завершить уже существующий.

Итератор `it` из последнего фрагмента кода не сообщает о завершении своей работы: когда вы получаете значение `3`, `false` меняется на `true`. Но чтобы узнать об этом, вам приходится еще раз вызывать метод `next()`, фактически выходя за границу значений массива. Чуть позже вы поймете, почему такое проектное решение, как правило, считается наилучшим вариантом.

Примитивные строковые значения также по умолчанию итерируемы:

```
var greeting = "hello world";

var it = greeting[Symbol.iterator]();

it.next();           // { value: "h", done: false }
it.next();           // { value: "e", done: false }
..
```



С технической точки зрения само по себе значение примитива итерируемым не является, но благодаря «упаковке» строка `"hello world"` преобразуется в итерируемый объект-оболочку `String`. Подробно данная тема рассматривается в книге *Types & Grammar* этой серии.

В ES6 также появилось несколько новых структур данных, называемых коллекциями (см. главу 5). Коллекции не только итерируемы, но и предоставляют методы API для генерации итератора. Например:

```
var m = new Map();
m.set( "foo", 42 );
m.set( { cool: true }, "hello world" );

var it1 = m[Symbol.iterator]();
var it2 = m.entries();

it1.next();           // { value: [ "foo", 42 ], done: false }
it2.next();           // { value: [ "foo", 42 ], done: false }
..
```

Метод итератора `next(..)` в зависимости от вашего желания принимает один или несколько аргументов. Встроенные итераторы эту возможность в основном не используют, чего нельзя сказать об итераторе генератора (см. ниже раздел «Генераторы»).

По общему соглашению, вызов метода `next(..)` после завершения работы итератора (в том числе для встроенных итераторов) не приводит к ошибке, а дает результат `{ value: undefined, done: true }`.

Необязательные методы: `return(..)` и `throw(..)`

Необязательные методы интерфейса итератора — `return(..)` и `throw(..)` — у большинства встроенных итераторов не реализуются. Тем не менее они, безусловно, значимы в контексте генераторов, поэтому более подробную информацию вы найдете в разделе «Генераторы».

Метод `return(..)` определен как отправляющий итератору сигнал о том, что код извлечения информации завершил работу и больше не будет извлекать значения. Этот сигнал уведомляет итератор, отвечающий на вызовы метода `next(..)`, что пришло время приступить к очистке, например к освобождению/закрытию сетевого соединения, базы данных или дескриптора файла.

Если у итератора есть метод `return(..)` и при этом возникает любое условие, которое можно интерпретировать как аномальное или раннее прекращение использования итератора, этот метод вызывается автоматически. Впрочем, это легко сделать и вручную.

Метод `return(..)`, как и `next(..)`, возвращает объект `IteratorResult`. В общем случае необязательное значение, отправленное в метод `return(..)`, будет возвращено как значение в объекте `IteratorResult`, хотя возможны и другие варианты развития событий.

Метод `throw(..)` сообщает итератору об исключении или ошибке, причем эту информацию тот может использовать иначе, нежели подаваемый методом `return(..)` сигнал завершения. Ведь, в отличие

от последнего, исключение или ошибка вовсе не подразумевает обязательного прекращения работы итератора.

Например, в случае с итераторами генератора метод `throw(..)`, по сути, вставляет в приостановленный контекст выполнения генератора порожденное им исключение, которое может быть перехвачено оператором `try...catch`. Необработанное исключение в конечном счете прерывает работу итератора.



По общему соглашению, после вызова методов `return(..)` или `throw(..)` итератор не должен больше генерировать никаких результатов.

Цикл итератора

Как было сказано в разделе «Цикл `for...of`» главы 2, появившийся в ES6 цикл `for...of` работает непосредственно с итерируемыми объектами.

Если итератор сам является итерируемым, его можно напрямую использовать с циклом `for...of`. Чтобы сделать итератор таковым, его следует передать методу `Symbol.iterator`, который вернет нужный результат:

```
var    it = {  
    // делаем итератор 'it' итерируемым  
    [Symbol.iterator]() { return this; },  
  
    next() { .. },  
    ..  
};  
  
it[Symbol.iterator]() === it; // true
```

Теперь можно вставить итератор `it` в цикл `for...of`:

```
for (var v of it) {  
    console.log( v );  
}
```

Чтобы полностью понять, что тут происходит, вспомните, как работает эквивалент цикла `for..of` — цикл `for` (мы говорили о нем в главе 2):

```
for (var v, res; (res = it.next()) && !res.done; ) {  
    v = res.value;  
    console.log( v );  
}
```

При ближайшем рассмотрении мы видим, что перед каждой итерацией вызывается метод `it.next()`, после чего происходит сравнение с переменной `res.done`. Если она имеет значение `true`, выражение получает результат `false`, и следующей итерации не происходит.

Напомню, что ранее мы уже говорили о принятой для итераторов тенденции не возвращать вместе с предполагаемым окончательным значением `done: true`. Теперь вы видите, почему так происходит.

Если итератор вернет `{ done: true, value: 42 }`, цикл `for..of` отбросит последнее значение 42, и оно будет потеряно. Именно из-за того, что итераторы могут использоваться такими шаблонами, как цикл `for..of` или его эквиваленты, следует подождать с возвращением сигнализирующего о завершении работы значения `done: true`, пока не будут возвращены все связанные с итерациями значения.



Разумеется, ничто не мешает целенаправленно спроектировать итератор, который будет одновременно возвращать последнее значение и `done: true`. Но такие вещи обязательно следует документировать, потому что в противном случае вы неявно заставите потребителей итератора использовать шаблон итераций, отличный от задаваемого циклом `for..of` или его эквивалентами.

Пользовательские итераторы

В дополнение к стандартным встроенным итераторам теперь вы можете создать собственный. Для обеспечения взаимодействия с элементами ES6, использующими итераторы (например, с циклом `for...of` или оператором `...`), достаточно корректного интерфейса (или интерфейсов).

Давайте создадим итератор, генерирующий бесконечную последовательность чисел Фибоначчи:

```
var    Fib = {
  [Symbol.iterator]() {
    var    n1 = 1, n2 = 1;

    return {
      // делаем итератор итерируемым
      [Symbol.iterator]() { return this; },

      next() {
        var current = n2;
        n2 = n1;
        n1 = n1 + current;
        return { value: current, done: false };
      },

      return(v) {
        console.log(
          "Последовательность Фибоначчи завершена."
        );
        return { value: v, done: true };
      }
    };
  }
};

for    (var v of Fib) {
  console.log( v );

  if    (v > 50) break;
}
// 1 1 2 3 5 8 13 21 34 55
// Последовательность Фибоначчи завершена
```



Без оператора `break` цикл `for...of` работал бы бесконечно, а это не имеет смысла.

Метод `Fib[Symbol.iterator]()` возвращает объект-итератор, обладающий методами `next()` и `return(...)`. Состояние поддерживается посредством переменных `n1` и `n2`, сохраненных замыканием.

А теперь рассмотрим итератор, выполняющий по очереди некоторые действия:

```
var tasks = {
  [Symbol.iterator]() {
    var steps = this.actions.slice();

    return {
      // делаем итератор итерируемым
      [Symbol.iterator]() { return this; },

      next(...args) {
        if (steps.length > 0) {
          let res = steps.shift()( ...args );
          return { value: res, done: false };
        }
        else {
          return { done: true }
        }
      },

      return(v) {
        steps.length = 0;
        return { value: v, done: true };
      }
    };
  },
  actions: []
};
```

Итератор объекта `tasks` перебирает функции, обнаруженные в свойстве-массиве `actions`, и выполняет по очереди, передавая в них аргументы, которые ранее были переданы методу `next(...)`,

а затем возвращая полученное значение в виде стандартного объекта `IteratorResult`.

Вот как мы можем воспользоваться этой очередью задач:

```
tasks.actions.push(  
  function step1(x){  
    console.log( "step 1:", x );  
    return x * 2;  
  },  
  function step2(x,y){  
    console.log( "step 2:", x, y );  
    return x + (y * 2);  
  },  
  function step3(x,y,z){  
    console.log( "step 3:", x, y, z );  
    return (x * y) + z;  
  }  
);  
  
var it = tasks[Symbol.iterator]();  
  
it.next( 10 );           // step 1: 10  
                        // { value: 20, done: false }  
  
it.next( 20, 50 );      // step 2: 20 50  
                        // { value: 120, done: false }  
  
it.next( 20, 50, 120 ); // step 3: 20 50 120  
                        // { value: 1120, done: false }  
  
it.next();              // { done: true }
```

Пример демонстрирует, что итераторы могут применяться в качестве шаблона не только для данных, но и для структурирующих алгоритмов. Мы еще поговорим об этом в следующем разделе при обсуждении генераторов.

Можно даже подойти к вопросу творчески и задать итератор, выполняющий метаоперации с одним фрагментом данных. Например, давайте определим итератор для чисел в диапазоне

от 0 до какого-то заданного положительного или отрицательного значения:

```
if (!Number.prototype[Symbol.iterator]) {
  Object.defineProperty(
    Number.prototype,
    Symbol.iterator,
    {
      writable: true,
      configurable: true,
      enumerable: false,
      value: function iterator(){
        var i, inc, done = false, top = +this;

        // итерации в положительную или отрицательную
        // сторону?
        inc = 1 * (top < 0 ? -1 : 1);

        return {
          // делаем итерируемым сам итератор
          [Symbol.iterator]() { return this; },

          next() {
            if (!done) {
              // начальная итерация всегда 0
              if (i == null) {
                i = 0;
              }
              // итерации в положительном
              // направлении
              else if (top >= 0) {
                i = Math.min(top, i + inc);
              }
              // итерации в отрицательном
              // направлении
              else {
                i = Math.max(top, i + inc);
              }

              // закончить после этой итерации?
              if (i == top) done = true;

              return { value: i, done: false };
            }
          }
        };
      }
    }
  );
}
```

```

    }
    else {
        return { done: true };
    }
}
};
}
);
}
}

```

Что же мы получили в результате такого творческого подхода?

```

for (var i of 3) {
    console.log( i );
}
// 0 1 2 3
[...-3];           // [0, -1, -2, -3]

```

Эти приемы кажутся забавными, хотя их практическая польза остается до некоторой степени спорной. В то же время может возникнуть вопрос: почему в ES6 отсутствует эта незначительная, но приятная функциональная особенность?

Было бы упущением не напомнить вам, что к расширению встроенных прототипов, пример одного из которых вы видели в последнем фрагменте кода, нужно подходить аккуратно и все время помнить о потенциальных опасностях.

В рассмотренном случае шансы конфликта с другим кодом или с какой-нибудь будущей функциональной особенностью JS, скорее всего, исчезающе малы. Но даже к такой малой вероятности следует подходить серьезно, поэтому подробно документируйте такие вещи, чтобы избежать проблем в будущем.



Я расширил эту технику, см. <http://blog.getify.com/iterating-es6-numbers/>. В одном из комментариев к записи <http://blog.getify.com/iterating-es6-numbers/comment-page-1/#comment-535294> мне предложили использовать аналогичный прием для диапазонов строковых символов.

Применение итераторов

Вы уже видели, как итератор шаг за шагом работает в цикле `for...of`. Но им могут пользоваться и другие структуры.

Рассмотрим итератор, присоединенный к массиву (хотя аналогичное поведение будет демонстрировать любой выбранный нами итератор):

```
var a = [1,2,3,4,5];
```

Оператор разделения `...` исчерпывает итератор полностью. Например:

```
function foo(x,y,z,w,p) {  
    console.log( x + y + z + w + p );  
}
```

```
foo( ...a );                // 15
```

Кроме того, можно распределить итератор внутри массива:

```
var b = [ 0, ...a, 6 ];  
b;                          // [0,1,2,3,4,5,6]
```

При процедуре деструктуризации массива (см. раздел «Деструктурирующее присваивание» главы 2) итератор может использоваться как частично, так и полностью (в сочетании с `rest/gather`-оператором `...`).

```
var it = a[Symbol.iterator]();  
  
var [x,y] = it;  
// берем из 'it' только первые два элемента  
var [z, ...w] = it;  
// берем третий элемент, а затем сразу все остальные  
  
// истощен ли 'it' полностью? Да  
it.next();                // { value: undefined, done: true }  
  
x;                          // 1
```

```
y;           // 2
z;           // 3
w;           // [4,5]
```

Генераторы

Все функции работают до своего завершения, верно? Другими словами, запущенная функция заканчивает работу до того, как начнет выполняться другая операция.

По крайней мере, именно так обстояли дела на протяжении всей истории существования языка JavaScript. Но в ES6 появилась новая, несколько необычная форма функции, названная генератором. Такая функция может остановиться во время выполнения, а затем продолжить работу с прерванного места.

Более того, каждый цикл остановки и возобновления работы позволяет передавать сообщения в обе стороны. Как генератор может вернуть значение, так и восстанавливающий его работу управляющий код может переслать туда что-нибудь.

Так же, как в случае с итераторами, рассмотренными в предыдущем разделе, на генераторы, вернее на то, для чего они в основном предназначены, можно смотреть с разных сторон. Единственно верной точки зрения не существует.



Более подробно генераторы рассматриваются в книге *Async & Performance* этой серии, а также в главе 4 данной книги.

Синтаксис

Функция-генератор объявляется следующим образом:

```
function *foo() {
  // ..
}
```

Положение звездочки `*` функционально несущественно. То же самое объявление можно написать любым из следующих способов:

```
function *foo() { .. }  
function* foo() { .. }  
function * foo() { .. }  
function*foo() { .. }  
..
```

Единственное, что имеет значение в данном случае, — это стилистические предпочтения. Авторы большинства руководств пишут `function* foo(..) { .. }`. Мне же больше нравится вариант `function *foo(..) { .. }` — его я и буду придерживаться в этой и следующих книгах.

Причина тут чисто дидактического характера. В тексте для обозначения функции-генератора я буду писать `*foo(..)`, в то время как обычная функция выглядит как `foo(..)`. Также символ `*` расположен и в объявлении `function *foo(..) { .. }`.

Кроме того, существует краткая форма генератора в объектных литералах (с краткими методами вы познакомились в главе 2):

```
var a = {  
  *foo() { .. }  
};
```

С моей точки зрения, в случае кратких генераторов запись `*foo() { .. }` выглядит естественнее, чем `*foo() { .. }`, — еще один довод, почему лучше писать так, как это делаю я. Единообразие в подобных вещах облегчает понимание материала.

Выполнение генератора

Хотя генератор объявляется с символом `*`, вызывается он как обычная функция:

```
foo();
```

В него можно передавать аргументы:

```
function *foo(x,y) {  
    // ..  
}  
  
foo( 5, 10 );
```

Основное отличие состоит в том, что запуск генератора, например `foo(5,10)`, не приводит к исполнению его кода. Вместо этого создается итератор, контролирующий то, как генератор исполняет свой код.

Мы вернемся к этому в разделе «Контроль со стороны итератора», а пока ограничимся примером:

```
function *foo() {  
    // ..  
}  
  
var it = foo();  
  
// чтобы начать/продолжить выполнение '*foo()',  
// вызываем 'it.next()'
```

Ключевое слово `yield`

Внутри генераторов используется ключевое слово, сигнализирующее о прерывании работы: `yield`. Рассмотрим пример:

```
function *foo() {  
    var x = 10;  
    var y = 20;  
  
    yield;  
  
    var z = x + y;  
}
```

В этом генераторе `*foo()` сначала запускаются операции из первых двух строк, а затем ключевое слово `yield` останавливает работу.

После ее возобновления запускается последняя строчка генератора `*foo()`. Ключевое слово `yield` может появляться в генераторе произвольное количество раз (или вообще отсутствовать).

Ничто не мешает поместить ключевое слово `yield` в тело цикла, создав повторяющуюся точку останова. При этом в случае с бесконечным циклом вы получите генератор, работа которого никогда не завершается, что иногда бывает вполне допустимо и даже необходимо.

Ключевое слово `yield` не просто прерывает работу генератора. В момент остановки оно посылает наружу значение. Вот пример цикла `while..true` внутри генератора, который на каждой итерации получает новое случайное число:

```
function *foo() {  
  while (true) {  
    yield Math.random();  
  }  
}
```

Выражение `yield ..` позволяет не только передать значение (ключевое слово `yield`, не сопровождающееся ничем, означает `yield undefined`), но и получить — то есть заместить — конечное значение при возобновлении работы генератора. Рассмотрим пример:

```
function *foo() {  
  var x = yield 10;  
  console.log( x );  
}
```

Этот генератор в момент остановки сначала получит значение 10. После возобновления работы — методом `it.next(..)` — любое восстановленное значение (если таковое вообще существует) полностью заместит выражение `yield 10`, то есть именно оно будет присвоено переменной `x`.

Выражение `yield ..` может появляться во всех тех местах, куда вставляют обычные выражения. Например:

```
function *foo() {  
  var arr = [ yield 1, yield 2, yield 3 ];  
  console.log( arr, yield 4 );  
}
```

В данном случае генератор `*foo()` содержит четыре записи `yield ...`. Каждая из них останавливает работу генератора и ожидает некоего нового значения, которое затем вставляется в различные контексты выражения.

Строго говоря, слово `yield` — не оператор, хотя выражения вида `yield 1` выглядят именно как операторы. Тем не менее оно используется и само по себе, например `var x = yield;`, поэтому его представление в качестве оператора может привести к путанице.

С технической точки зрения выражение `yield ..` имеет такой же приоритет (концептуально в данном случае это соответствует приоритету операторов), как и выражение присваивания, например `a = 3`. Таким образом, `yield ..` может появляться везде, где допустимо выражение `a = 3`.

Проиллюстрируем эту аналогию:

```
var a, b;  
  
a = 3;           // valid  
b = 2 + a = 3;   // invalid  
b = 2 + (a = 3); // valid  
  
yield 3;         // valid  
a = 2 + yield 3; // invalid  
a = 2 + (yield 3); // valid
```



Если как следует подумать, то в одинаковом поведении выражения `yield ..` и выражения присваивания есть концептуальный смысл. Когда после остановки генератор возобновляет работу, выражение с ключевым словом `yield` завершается/замещается значением восстановления, что фактически аналогично присваиванию последнего.

Если нужно вставить `yield ..` в место, где недопустимы такие выражения, как `a = 3`, его помещают в скобки ().

Из-за низкого приоритета ключевого слова `yield` практически все выражения, следующие за `yield ..`, будут вычисляться раньше. Более низким приоритетом обладают только оператор распределения `...` и запятая `,`, до них дело доходит после вычисления выражения с `yield`.

Как и в случае с набором обычных операторов, есть еще один вариант применения скобок `()`. Они позволяют переопределить (поднять) низкий приоритет ключевого слова `yield`, как в случае с этими двумя выражениями:

```
yield 2 + 3;           // то же самое, что и 'yield (2 + 3)'  
(yield 2) + 3;        // сначала 'yield 2', затем '+ 3'
```

Подобно оператору присваивания `=`, ключевое слово `yield` имеет правую ассоциативность. Это означает, что набор последовательных выражений `yield` рассматривается как сгруппированный с помощью скобок `(..)` справа налево. То есть `yield yield yield 3` трактуется как `yield (yield (yield 3))`. Трактовка с левой ассоциативностью, то есть `((yield) yield) yield 3` не имеет смысла.

При комбинации ключевого слова `yield` с какими-либо операторами или с другими ключевыми словами `yield` разумно прибегнуть к группировке с помощью скобок `(..)`, чтобы четко обозначить свои намерения (как вы помните, с другими операторами дело обстоит так же).



Подробнее тема приоритета операторов и ассоциативность рассматриваются в книге *Types & Grammar* этой серии.

Выражение `yield *`

Аналогично тому, как символ `*` превращает объявление функции в объявление генератора, в случае использования `*` с ключевым словом `yield` образуется совершенно другой механизм, называемый

yield-делегированием. Грамматически выражение `yield *..` будет вести себя так же, как и рассмотренное в предыдущем разделе `yield ...`.

Выражению `yield *..` требуется итерируемый объект; оно вызывает его итератор и передает ему управление собственным генератором. Рассмотрим пример:

```
function *foo() {  
    yield *[1,2,3];  
}
```



Как и в случае с объявлением генератора, положение символа `*` в выражениях `yield *` зависит только от ваших стилистических предпочтений. В других литературных источниках предпочитают запись `yield* ..`, я же выбрал вариант `yield *..` по причинам, рассматривавшимся в предыдущем разделе.

Значение `[1,2,3]` даст нам итератор, который будет пошагово передавать свои значения генератору `*foo()`. Другой способ продемонстрировать такое поведение — сделать `yield-делегирование` другому генератору:

```
function *foo() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
function *bar() {  
    yield *foo();  
}
```

Итератор, появившийся при вызове генератором `*bar()` генератора `*foo()`, делегируется при помощи выражения `yield *`. Это означает, что все значения, порождаемые генератором `*foo()`, будут выводиться генератором `*bar()`.

Если завершающее значение выражения `yield ..` появляется при возобновлении работы генератора методом `it.next(..)`, то завершающее значение выражения `yield *..` представляет собой значение, возвращаемое итератором, которому были делегированы полномочия (если таковое существует).

У встроенных итераторов возвращаемые значения, как правило, отсутствуют, как было показано в конце раздела «Цикл итератора». Но собственноручно созданный вами итератор (или генератор) можно заставить возвращать значение, которым в конечном итоге и воспользуется выражение `yield *..`:

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
  return 4;
}

function *bar() {
  var x = yield *foo();
  console.log( "x:", x );
}

for (var v of bar()) {
  console.log( v );
}
// 1 2 3
// x: 4
```

Если значения 1, 2 и 3 отданы генератором `*foo()`, а затем генератором `*bar()`, то возвращенное генератором `*foo()` значение 4 является завершающим для выражения `yield *foo()` и присваивается переменной `x`.

Выражение `yield *` может не только вызвать еще один генератор (путем делегирования его итератору), но и породить рекурсию генератора, вызывая само себя:

```
function *foo(x) {  
    if (x < 3) {  
        x = yield *foo( x + 1 );  
    }  
    return x * 2;  
}  
  
foo( 1 );
```

В результате `foo(1)` и последующего вызова метода `next()` итератора для прохода через этапы рекурсии мы получим 24. При первом запуске генератора `*foo(..)` переменная `x` имеет значение 1, то есть соблюдается условие `x < 3`. Выражение `x + 1` рекурсивно передается генератору `*foo(..)`, соответственно, `x` приобретает значение 2. Следующий рекурсивный вызов дает переменной `x` значение 3.

После этого условие `x < 3` перестает выполняться, и рекурсия прекращается, а оператор `return 3 * 2` возвращает выражению `yield *..` из предыдущего вызова значение 6, которое, в свою очередь, присваивается переменной `x`. Следующий оператор `return 6 * 2` возвращает предыдущему вызову `x` значение 12. Наконец, после завершающего этапа работы генератора `*foo(..)` возвращается значение `12 * 2`, то есть 24.

Контроль со стороны итератора

Выше уже упоминалось, что генераторы управляются итераторами. Давайте подробно разберем этот процесс.

Для примера рассмотрим рекурсивную форму генератора `*foo(..)` из предыдущего раздела. Вот как она действует:

```
function *foo(x) {  
    if (x < 3) {  
        x = yield *foo( x + 1 );  
    }  
    return x * 2;  
}
```

```
}  
  
var it = foo( 1 );  
it.next();           // { value: 24, done: true }
```

В этом случае генератор вообще не останавливает свою работу, так как выражение `yield ..` отсутствует. Вместо него в коде имеется выражение `yield *`, которое обеспечивает прохождение каждой итерации путем рекурсивного вызова. Так что работа генератора осуществляется исключительно вызовами функции `next()` итератора.

Теперь рассмотрим генератор, имеющий несколько этапов и соответственно дающий несколько значений:

```
function *foo() {  
    yield 1;  
    yield 2;  
    yield 3;  
}
```

Мы уже знаем, что воспользоваться итератором, даже если тот присоединен к генератору `*foo()`, можно с помощью цикла `for..of`:

```
for (var v of foo()) {  
    console.log( v );  
}  
// 1 2 3
```



Для цикла `for..of` необходим итерируемый объект. Сама по себе ссылка на функцию генератора (например, `foo`) таковым считаться не может; для получения итератора ее следует выполнить как `foo()` (при этом, как говорилось выше, такой итератор сам является итерируемым). Теоретически можно расширить `GeneratorPrototype` (прототип всех функций генератора) функцией `Symbol.iterator`, которая, по сути, всего лишь возвращает `this()`. Это сделает ссылку `foo` итерируемой, что обеспечит работу цикла `for (var v of foo) { .. }` (обратите внимание на отсутствие скобок `()` у функции `foo`).

Попробуем выполнить итерации генератора вручную:

```
function *foo() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
var it = foo();  
  
it.next();           // { value: 1, done: false }  
it.next();           // { value: 2, done: false }  
it.next();           // { value: 3, done: false }  
  
it.next();           // { value: undefined, done: true }
```

Мы видим, что на три оператора `yield` приходится четыре вызова метода `next()`. Такое несовпадение может показаться странным. Однако количество вызовов метода `next()` всегда на единицу превышает количество выражений `yield`, тем самым давая возможность сделать все вычисления и позволить генератору отработать до конца.

Впрочем, если смотреть с другой стороны (изнутри, а не снаружи), более осмысленным кажется равное число `yield` и `next()`.

Напомню, что выражение `yield ..` завершается значением, которое используется в момент возобновления работы генератора. Таким образом, передаваемый в метод `next(..)` аргумент завершает выражение, приостановленное в текущий момент.

Проиллюстрируем это следующим фрагментом кода:

```
function *foo() {  
    var x = yield 1;  
    var y = yield 2;  
    var z = yield 3;  
    console.log( x, y, z );  
}
```


Теперь, когда стало видно, что на каждый вопрос выражения `yield ..` отвечает следующий вызов метода `next(..)`, легко понять, что наблюдаемый нами «лишний» вызов метода `next()` — это самая первая операция, которая запускает работу генератора.

Объединим все шаги:

```
var it = foo();

// запускает генератор
it.next();                // { value: 1, done: false }

// отвечает на первый вопрос
it.next( "foo" );         // { value: 2, done: false }

// отвечает на второй вопрос
it.next( "bar" );         // { value: 3, done: false }

// отвечает на третий вопрос
it.next( "baz" );         // "foo" "bar" "baz"
                        // { value: undefined, done: true }
```

Генератор можно представить как поставщик значений, в этом случае каждая итерация просто создает значение для дальнейшего использования.

Но в более общем смысле, возможно, корректнее будет рассматривать генераторы как управляемое поэтапное выполнение кода, во многом напоминающее очередь задач, рассмотренную в разделе «Пользовательские итераторы».



Изложенная выше точка зрения — это повод снова вернуться к рассмотрению генераторов в главе 4. В частности, нет поводов вызывать очередной метод `next(..)` сразу же после завершения работы предыдущего. В то время, когда внутренний контекст генератора приостановлен, прочие части программы продолжают выполняться, предоставляя в числе прочего возможность для управления асинхронными действиями при возобновлении работы генератора.

Раннее завершение

Как уже упоминалось, присоединенный к генератору итератор поддерживает необязательные методы `return(...)` и `throw(...)` — оба немедленно прерывают работу приостановленного генератора.

Рассмотрим пример:

```
function *foo() {  
    yield 1;  
    yield 2;  
    yield 3;  
}  
  
var it = foo();  
  
it.next();           // { value: 1, done: false }  
  
it.return( 42 );     // { value: 42, done: true }  
  
it.next();           // { value: undefined, done: true }
```

Метод `return(x)` как бы вынуждает немедленно выполнить оператор `return x`, и вы сразу получаете указанное значение. Генератор, чья работа была завершена обычным образом или преждевременно, как показано в примере, больше ничего не делает ни с каким кодом и не возвращает никаких значений.

Метод `return(...)` может вызываться не только вручную, но и автоматически в конце итераций. В последнем случае его вызывает компонент, использующий итератор, например цикл `for...of` или оператор разделения `....`.

Эта функциональная особенность позволяет уведомить генератор, что контролирующий код прекратил выполнять перебор и можно приступить к задачам, связанным с очисткой (освобождением ресурсов, сбросом состояния и т. п.). Данная задача, аналогично обычному шаблону очистки функции, в основном решается с помощью оператора `finally`:

```
function *foo() {
  try {
    yield 1;
    yield 2;
    yield 3;
  }
  finally {
    console.log( "cleanup!" );
  }
}

for (var v of foo()) {
  console.log( v );
}
// 1 2 3
// очистка!

var it = foo();

it.next();           // { value: 1, done: false }
it.return( 42 );     // очистка!
                    // { value: 42, done: true }
```



Помещать оператор `yield` внутрь оператора `finally` нельзя! Формально такое допустимо, но делать этого не стоит. В определенном смысле `finally` действует как отложенное завершение вызванного вами метода `return(..)`, потому что любое выражение `yield ..` в операторе `finally` будет прерывать его работу и отправлять сообщения; вы не получите немедленно завершающийся генератор, как ожидаете. Нет разумных причин использовать настолько сумасшедший вариант, поэтому избегайте его!

Представленный выше фрагмент не только показывает, как метод `return(..)` прерывает работу генератора, запуская оператор `finally`, но и демонстрирует, что при каждом вызове генератор формирует новый итератор. Более того, вы можете использовать несколько итераторов, одновременно присоединенных к одному и тому же генератору:

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
}

var it1 = foo();
it1.next();           // { value: 1, done: false }
it1.next();           // { value: 2, done: false }

var it2 = foo();
it2.next();           // { value: 1, done: false }

it1.next();           // { value: 3, done: false }

it2.next();           // { value: 2, done: false }
it2.next();           // { value: 3, done: false }

it2.next();           // { value: undefined, done: true }
it1.next();           // { value: undefined, done: true }
```

Раннее прерывание

Вместо метода `return(...)` можно вызвать метод `throw(...)`. Аналогично тому, как метод `return(x)`, по сути, представляет собой оператор `return x`, вставленный в текущую точку останова генератора, вызов метода `throw(x)` вставляет туда оператор `throw x`.

Помимо генерации исключений (операторы `try` будут рассмотрены в следующем разделе), метод `throw(...)` выполняет раннее завершение, прерывая работу генератора в текущей точке останова. Например:

```
function *foo() {
  yield 1;
  yield 2;
  yield 3;
}

var it = foo();

it.next();             // { value: 1, done: false }
```

```
try {  
    it.throw( "Oops!" );  
}  
catch (err) {  
    console.log( err );    // Исключение: Ой!  
}  
  
it.next();                // { value: undefined, done: true }
```

Так как метод `throw(..)`, по сути, вставляет вместо строчки `yield 1` оператор `throw ..`, и при этом обработчик исключения отсутствует, исключение немедленно возвращается к вызывающему коду, который использует блок `try..catch`.

В отличие от метода `return(..)`, метод итератора `throw(..)` автоматически никогда не вызывается.

В представленном фрагменте кода этого не показано, но если блок `try..finally` в момент вызова метода `throw(..)` ожидал внутри генератора, оператор `finally` получает шанс завершить работу до момента, когда исключение вернется в вызывающий код.

Обработка ошибок

Я уже упоминал, что в случае с генераторами обработка ошибок реализуется при помощи блока `try..catch`, который работает как во входящем, так и в исходящем направлении.

```
function *foo() {  
    try {  
        yield 1;  
    }  
    catch (err) {  
        console.log( err );  
    }  
  
    yield 2;  
  
    throw "Hello!";  
}
```

```
}

var it = foo();

it.next();                // { value: 1, done: false }

try {
  it.throw( "Hi!" );      // Hi!
                          // { value: 2, done: false }
  it.next();

  console.log( "never gets here" );
}
catch (err) {
  console.log( err );     // Hello!
}
```

Ошибки также могут распространяться в обоих направлениях через делегат `yield *`:

```
function *foo() {
  try {
    yield 1;
  }
  catch (err) {
    console.log( err );
  }

  yield 2;

  throw "foo: e2";
}

function *bar() {
  try {
    yield *foo();

    console.log( "never gets here" );
  }
  catch (err) {
    console.log( err );
  }
}
```

```
}

var it = bar();

try {
    it.next();           // { value: 1, done: false }
    it.throw( "e1" );    // e1
                        // { value: 2, done: false }

    it.next();           // foo: e2
                        // { value: undefined, done: true }
}
catch (err) {
    console.log( "never gets here" );
}

it.next();              // { value: undefined, done: true }
```

Вы уже видели, что когда генератор `*foo()` вызывает оператор `yield 1`, значение 1 проходит через генератор `*bar()` без изменений.

Впрочем, самое интересное в этом фрагменте то, что при вызове генератором `*foo()` строки `throw "foo: e2"` ошибка переходит в генератор `*bar()` и немедленно перехватывается вставленным туда блоком `try...catch`. Ей не удастся беспрепятственно пройти через `*bar()`, как это делает значение 1.

Оператор `catch` внутри генератора `*bar()` выполняет обычный вывод `err ("foo: e2")`, после чего генератор обычным образом завершает свою работу. Вот почему от метода `it.next()` приходит результат итератора `{ value: undefined, done: true }`.

Если бы в `*bar()` возле выражения `yield *..` отсутствовал блок `try...catch`, ошибка, разумеется, проходила бы генератор насквозь, и при этом все равно требовалось бы завершить его работу.

Транскомпиляция генератора

Можно ли было представить что-нибудь подобное генератору в JS до появления стандарта ES6? Оказывается, генераторы существо-

вали и в более ранних версиях, мало того, есть несколько хороших инструментов для их реализации, в том числе такие серьезные, как Regenerator от Facebook (<https://facebook.github.io/regenerator/>).

Чтобы лучше понять, что представляют собой генераторы, давайте попробуем выполнить преобразование вручную. По сути, мы собираемся создать простой конечный автомат на базе замыкания.

Исходный генератор будет очень простым:

```
function *foo() {  
  var x = yield 42;  
  console.log( x );  
}
```

Для начала нам потребуется функция `foo()`, которую мы можем выполнять и которая должна возвращать итератор.

```
function foo() {  
  // ..  
  
  return {  
    next: function(v) {  
      // ..  
    }  
  
    // пропустим методы 'return(..)' и 'throw(..)'  
  };  
}
```

Теперь нужна внутренняя переменная для отслеживания нашего положения в логической схеме этого «генератора». Назовем ее `state`. У нас будет три состояния: 0 — изначальное, 1 — в ожидании завершения выражения `yield` и 2 — после завершения работы генератора.

Нам нужно, чтобы при каждом вызове метода `next(..)` обрабатывался следующий шаг и происходило инкрементирование состояния. Для удобства мы поместим каждый шаг в оператор `case` блока

`switch`, и все это будет расположено в теле внешней функции `nextState(..)`, доступной для вызова методу `next(..)`. Так как переменная `x` находится в области видимости нашего «генератора», ее следует поместить за пределами функции `nextState(..)`.

Вот как все это будет выглядеть (разумеется, в упрощенном виде, позволяющем наглядно продемонстрировать принцип преобразования):

```
function foo() {
  function nextState(v) {
    switch (state) {
      case 0:
        state++;

        // выражение 'yield'
        return 42;
      case 1:
        state++;

        // выражение 'yield' выполнено
        x = v;
        console.log( x );

        // неявный оператор 'return'
        return undefined;

        // не нужно обрабатывать состояние '2'
    }
  }
  var state = 0, x;

  return {
    next: function(v) {
      var ret = nextState( v );

      return { value: ret, done: (state == 2) };
    }

    // пропустим методы 'return(..)' и 'throw(..)'
  };
}
```


Теперь давайте протестируем наш «генератор», работающий в стандартах, предшествующих ES6:

```
var it = foo();

it.next();           // { value: 42, done: false }

it.next( 10 );       // 10
                     // { value: undefined, done: true }
```

Неплохо, не так ли? Я надеюсь, упражнение позволило вам осознать, что генераторы — это всего лишь простой синтаксис для логической схемы конечного автомата, что и делает их широко применимыми.

Применение генераторов

Теперь, когда вы хорошо понимаете, как работают генераторы, поговорим о том, зачем они нужны.

Как правило, они используются в одном из двух следующих случаев.

Создание набора значений

Этот вариант применения может быть как простым (например, случайные строки или возрастающие числа), так и дающим более структурированный доступ к данным (например, итерации по строкам, которые вернул запрос к базе данных).

В любом случае мы контролируем генератор с помощью итератора, что дает возможность реализовывать некую логическую схему при каждом вызове метода `next(..)`. Обычные итераторы, работающие со структурами данных, просто извлекают оттуда значения, не добавляя никакой управляющей логической схемы.

Очередь задач для последовательного выполнения

Этот вариант применения часто представляет собой управление шагами какого-нибудь алгоритма, причем на каждом из них требуется извлекать данные из некоего внешнего источника — немедленно или с асинхронной задержкой.

С точки зрения кода внутри генератора детали реализации синхронного или асинхронного извлечения данных в точке оператора `yield` совершенно непрозрачны. Более того, они намеренно сделаны абстрактными, чтобы не прятать естественное последовательное выражение шагов за сложностями реализации. Кроме того, абстрагирование дает возможность часто менять или перерабатывать реализацию, никак не затрагивая код внутри генератора.

Если смотреть на генераторы с точки зрения их применения, они перестают быть всего лишь красивым синтаксисом для конечного автомата с ручным управлением. Генераторы — это мощный инструмент абстрагирования для систематизации и контроля за упорядоченным созданием и использованием данных.

Модули

Не будет преувеличением предположить, что наиболее важным способом структуризации кода в JavaScript всегда считался модуль. Для меня и едва ли не для всего сообщества разработчиков шаблон «модуль» обуславливает большую часть кода.

Старый способ

Традиционно шаблон «модуль» базировался на внешней функции с внутренними переменными и функциями, возвращающей «открытый API» с методами, которые замыкались на внутренних данных и функциональных особенностях. Зачастую это выражалось следующим образом:

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

  // открытый API
  return {
    greeting: greeting
  };
}

var me = Hello( "Kyle" );
me.greeting();                // Hello Kyle!
```

Модуль `Hello(...)` при вызове несколько раз подряд позволяет генерировать набор экземпляров. Иногда модуль необходим как синглтон (то есть нужен всего один экземпляр). В этом случае повсеместно применяется вариация предыдущего фрагмента с использованием IIFE:

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

  // открытый API
  return {
    greeting: greeting
  };
})( "Kyle" );

me.greeting();                // Hello Kyle!
```

Это проверенный временем шаблон. Он достаточно гибок и допускает широкий диапазон вариаций, подходящих к различным сценариям.

Наиболее распространены асинхронное (AMD — asynchronous module definition) и универсальное определения модуля (UMD — universal module definition). Детали реализации этих шаблонов

и техник мы рассматривать не будем, так как они подробно описаны в многочисленных источниках, доступных в Сети.

Продвигаясь вперед

В ES6 модули получили первоклассную синтаксическую и функциональную поддержку, для их реализации уже не требуется опираться на внешнюю функцию и замыкание.

Но перед тем как мы углубимся в детали синтаксиса, нужно понять несколько крайне важных различий между модулями в ES6 и в предыдущих версиях языка.

- Стандарт ES6 использует модули на основе файлов: один модуль содержится в одном файле. На текущий момент стандартизованный способ объединения в один файл нескольких модулей отсутствует.

Все это означает, что в веб-приложение в браузере их потребуется загружать по отдельности, а не в виде большой совокупности внутри единого файла, как было принято ранее для оптимизации производительности.

Ожидается, что HTTP/2 значительно смягчит проблемы с производительностью, так как он поддерживает постоянное соединение через сокет и позволяет оперативно загружать много небольших файлов как параллельно, так и один за другим.

- Модули ES6 обладают статическим API: вы статически определяете все виды экспорта верхнего уровня на открытом API вашего модуля, и изменить это потом уже будет нельзя.

Есть варианты применения, адаптированные к динамическому API, методы которого можно добавлять/удалять/менять, подстраиваясь под ситуацию в исполняемом коде. Вам придется или менять эти варианты, подгоняя их под стати-

ческие API в ES6, или ограничивать динамические изменения свойствами/методами объекта второго уровня.

- Модули ES6 являются синглтонами: существует только один экземпляр, поддерживающий состояние модуля. Каждый раз, импортируя его в другой модуль, вы получаете ссылку на единственный экземпляр. Для получения набора экземпляров модуль должен предоставить своего рода фабрику.
- Свойства и методы, доступные через открытый API модуля, нельзя рассматривать как обычные присваивания значений и ссылки. На самом деле это привязки (почти как указатели) к идентификаторам во внутреннем определении модуля.

В случае с модулями, которые существовали до ES6, если свойство, содержащее примитивное значение (например, число или строку), помещалось в открытый API, присваивание этого свойства выполнялось копированием значения, а все внутренние обновления соответствующей переменной производились отдельно и не влияли на открытую копию в объекте API.

В ES6 при экспорте локальной закрытой переменной, даже если в ней в этот момент хранится примитивная строка, число или что-нибудь подобное, экспортируется привязка к переменной. Если модуль меняет значение переменной, внешняя привязка импорта получает это новое значение.

- Импорт модуля аналогичен статическому запросу на его загрузку (если она еще не сделана). В браузере это предполагает блокирующую загрузку. Если вы находитесь на сервере (например, Node.js), аналогичная загрузка будет осуществляться из файловой системы.

Не стоит паниковать по поводу проблем с производительностью. Благодаря тому что у модулей ES6 есть статические определения, требования к импорту могут быть статически

просканированы, и загрузка произойдет заранее, до того как вы воспользуетесь модулем.

Однако ES6 не определяет и не обрабатывает механизм работы этих запросов на загрузку. Существует отдельное понятие загрузчика модулей, и в каждой среде выполнения (браузер, Node.js и т. п.) есть свой загрузчик по умолчанию. Для импорта модуля используется строковое значение, указывающее, где его можно получить (адрес URL, путь к файлу и т. п.), но в программе оно непрозрачно и имеет смысл только для загрузчика.

Вы можете определить собственный загрузчик, если вам нужен более полный контроль над процессом, чем тот, который предоставляет загрузчик по умолчанию, — в последнем случае контроль, по сути, отсутствует, так как все настройки полностью скрыты от кода программы.

Как видите, модули ES6 служат общей цели структуризации кода посредством инкапсуляции, контроля открытых API и ссылок на импорт зависимостей. Однако все это реализуется особым способом, скорее всего, непривычным для вас.

Модули CommonJS

Существует похожий, но не полностью совместимый со стандартом синтаксис модулей, который называется CommonJS. Он знаком тем, кто работает в экосистеме Node.js.

Предполагается, что в долгосрочной перспективе модули ES6 заместят собой все предшествующие форматы и стандарты, даже CommonJS, так как базируются на синтаксической поддержке в языке. Со временем этот подход неизбежно станет преобладающим, хотя бы по причине своего широкого распространения.

Однако то положение дел, которое мы имеем сейчас, складывалось долгое время. Существует добрая сотня тысяч модулей CommonJS

на серверной стороне и в десять раз больше модулей в различных форматах (UMD, AMD и пр.) на стороне браузера. Поэтому значительных изменений стоит ждать лишь по прошествии многих лет.

А пока, на нынешней переходной стадии, нам не обойтись без транскомпиляторов/преобразователей для модулей. Я допускаю, что вы уже привыкли к этой новой реальности и, будучи автором обычных модулей, AMD, UMD, CommonJS или ES6, учитываете тот факт, что они должны уметь анализировать среду, в которой им предстоит запускаться, и преобразовываться в подходящий формат.

Для среды Node.js это, скорее всего, означает CommonJS, для браузеров — UMD или AMD. В ближайшие несколько лет следует ожидать появления множества новых модулей, так как будет происходить их совершенствование и поиск наилучших подходов.

Поэтому вот лучшее, что я могу вам сейчас посоветовать: к какому бы формату вы ни были привязаны, начинайте изучать модули ES6 и воспринимать их как есть. Пусть все прочие тенденции постепенно сойдут на нет. Эти модули — будущее JS, даже если реальность пойдет немного по другому пути.

Новый способ

Два основных ключевых слова, активирующие модули ES6, — `import` и `export`. Этот синтаксис имеет множество нюансов, потому рассмотрим его более подробно.



Эту важную деталь легко упустить из виду: ключевые слова `import` и `export` всегда должны появляться на верхнем уровне области видимости, в которой будут применяться. Например, их нельзя вставлять в условный оператор `if`; они должны располагаться вне блоков и функций.

Экспорт членов API

Ключевое слово `export` или помещается перед объявлением, или используется в качестве своего рода оператора со специальным списком привязок, предназначенных для экспорта.

Например:

```
export function foo() {  
    // ..  
}  
  
export var awesome = 42;  
  
var bar = [1,2,3];  
export { bar };
```

Другой способ реализации этого же экспорта:

```
function foo() {  
    // ..  
}  
  
var awesome = 42;  
var bar = [1,2,3];  
  
export { foo, awesome, bar };
```

Все это называется *именованным экспортом* (named exports), так как вы, по сути, экспортируете привязки имени переменных, функций и пр.

Все, что вы не *помечаете* ключевым словом `export`, остается закрытым внутри области видимости модуля. То есть, несмотря на то что запись `var bar = ..` выглядит как объявление на верхнем уровне глобальной области видимости, на самом деле верхний уровень здесь — сам модуль; в модулях глобальной области видимости просто нет.



В действительности у модулей *есть* доступ к объекту `window` и всем связанным с ним «глобальным» вещам, но не как к лексической области видимости верхнего уровня. Тем не менее следует прилагать все усилия, чтобы в модулях не было никаких глобальных переменных.

В процессе именованного экспорта можно переименовать член модуля (это явление известно как псевдоним):

```
function foo() { .. }  
  
export { foo as bar };
```

Здесь при импорте доступным будет только член `bar`, а член `foo` останется скрытым внутри модуля.

Экспорт модулей отличается от обычного присваивания значений и ссылок, которое вы привыкли выполнять при помощи оператора `=`. На самом деле при экспорте какого-либо элемента вы передаете привязку (своего рода указатель).

Если вы поменяете внутри модуля значение переменной, привязку к которой уже экспортировали, то, даже если ее импорт уже завершился (см. следующий раздел), импортированная привязка даст текущее (обновленное) значение.

Рассмотрим пример:

```
var awesome = 42;  
export { awesome };  
  
// позднее  
awesome = 100;
```

Когда модуль уже импортирован, не имеет значения, до или после этого было выполнено присваивание `awesome = 100`. По завершении импортированная привязка начнет давать нам значение 100, а не 42.

Она ведь, по сути, представляет собой ссылку или указатель на саму переменную `awesome`, а не на копию ее значения. Эта концепция, появившаяся с введением привязок модулей в ES6, — одна из самых примечательных в JS.

Хотя внутри определения модуля ничто не запрещает несколько раз использовать ключевое слово `export`, в ES6 более предпочтителен подход, при котором оно существует в единственном экземпляре. Это так называемый *экспорт по умолчанию* (default export). По мнению некоторых членов комитета TC39, если следовать этому шаблону, можно получить более простой синтаксис.

Экспорт по умолчанию задает конкретную экспортированную привязку как вариант по умолчанию, выбираемый при импорте модуля. Вот почему этой привязке дали имя `default`. Позднее вы увидите, что во время импорта привязок модуля их можно переименовывать, и будете часто делать это в случае экспорта по умолчанию.

Возможна только одна привязка `default` на одно определение модуля. Импорт мы будем рассматривать в следующем разделе, и вы убедитесь, насколько сокращается синтаксис в случае экспорта модуля по умолчанию.

Здесь есть нюанс, на который следует обратить внимание. Давайте сравним два фрагмента. Первый:

```
function foo(..) {  
    // ..  
}  
  
export default foo;
```

Второй:

```
function foo(..) {  
    // ..  
}  
  
export { foo as default };
```

В первом случае вы экспортируете привязку в значение функционального выражения, а не в идентификатор `foo`. Другими словами, `export default ..` преобразовывает выражение. Если позднее присвоить функцию `foo` другому значению внутри модуля, процедура импорта все равно будет показывать изначально экспортированную функцию.

Кстати, первый фрагмент можно переписать еще и таким образом:

```
export default function foo(..) {  
  // ..  
}
```



Хотя с технической точки зрения `function foo..` в данном случае — функциональное выражение, в контексте внутренней области видимости модуля эта запись рассматривается как объявление функции, в котором имя `foo` связано с верхним уровнем области видимости модуля (это часто называют «поднятием»). То же самое верно для записи `export default class Foo...` Но если написать `export var foo = ..` вы можете, то `export default var foo = ..` (или `let`, или `const`) — нет, что являет собой крайне огорчительный пример несогласованности. На момент написания книги это уже обсуждается, и, возможно, ситуация будет исправлена в следующей версии стандарта JS.

Вернемся ко второму фрагменту:

```
function foo(..) {  
  // ..  
}  
  
export { foo as default };
```

Здесь привязка экспорта по умолчанию на самом деле выполняется к идентификатору `foo`, а не к его значению, то есть вы получаете ранее описанное поведение (в частности, если позднее вы поменяете значение `foo`, обновится и значение на стороне импорта).

Будьте крайне осторожны с этим небольшим подводным камнем в синтаксисе экспорта по умолчанию, особенно если логическая схема требует обновлять экспортируемые значения. Если вы не планируете этого делать, прекрасно подойдет вариант `export default ...`. В противном случае следует пользоваться вариантом `export { .. as default }`. И не забудьте снабдить ваш код комментариями, поясняющими ваши намерения!

Так как внутри модуля должно быть только одно ключевое слово `default`, возникает соблазн создать модуль с одним экспортом по умолчанию и экспортировать объект со всеми вашими методами API, например:

```
export default {  
  foo() { .. },  
  bar() { .. },  
  ..  
};
```

Такой шаблон практически совпадает со способом, которым разработчики структурировали модули до появления ES6, и потому кажется естественным. К сожалению, он имеет ряд недостатков, и официально его использовать не рекомендуется.

В частности, движок JS не умеет статически анализировать содержимое простого объекта, а это значит, что производительность статического импорта не будет оптимизироваться. Преимущество отдельного и явного экспорта каждого члена заключается как раз в том, что движок может сделать статический анализ и оптимизацию.

Если ваш API уже содержит несколько членов, возникает впечатление, что перечисленные принципы (один экспорт по умолчанию на один модуль, все члены API подвергаются именованному экспорту) — взаимоисключающие. Однако наличие единственной процедуры экспорта по умолчанию не исключает вариантов именованного экспорта.

Такой шаблон использовать не рекомендуется:

```
export default function foo() { .. }
```

```
foo.bar = function() { .. };  
foo.baz = function() { .. };
```

Следует написать вот что:

```
export default function foo() { .. }
```

```
export function bar() { .. }  
export function baz() { .. }
```



В предыдущем фрагменте имя `foo` использовалось для функции, помеченной словом `default`. Однако в контексте экспорта имя `foo` игнорировалось — экспортировано было имя `default`. При импорте этой привязки по умолчанию ей можно дать любое имя по вашему выбору, в чем вы убедитесь, читая следующий раздел.

В качестве альтернативы некоторые предпочитают такой вариант:

```
function foo() { .. }  
function bar() { .. }  
function baz() { .. }  
  
export { foo as default, bar, baz, .. };
```

Эффект комбинации экспорта по умолчанию с именованным экспортом станет более понятным, когда мы начнем рассматривать импорт. Но, по сути, это означает, что самая краткая форма импорта по умолчанию будет извлекать только функцию `foo()`. Пользователь может дополнительно вручную указать в качестве элементов именованного импорта функции `bar` и `baz`, если они ему нужны.

Представьте себе, сколь утомительным будет использование модуля при наличии многочисленных именованных привязок экспорта! Существуют шаблон импорта с символом обобщения, в котором все, что было экспортировано из модуля, импортируется внутри единого объекта пространства имен, но подобный импорт невозможен в привязки верхнего уровня.

Мы снова возвращаемся к тому, что данный механизм в ES6 специально разрабатывался таким образом, чтобы затруднить использо-

вание модулей с большим количеством экспортируемых элементов и поощрить проектирование простых модулей.

Я рекомендую избегать комбинаций экспорта по умолчанию и именованного экспорта, особенно при наличии большого API и ситуации, когда реструктуризация кода с целью разделения модулей непрактична или нежелательна. В этом случае достаточно воспользоваться именованным экспортом и документировать тот факт, что пользователи вашего модуля должны придерживаться подхода `import * as ..` (о нем пойдет речь в следующем разделе), чтобы импортировать API целиком в едином пространстве имен.

Выше это уже упоминалось, но давайте рассмотрим этот момент еще раз, более детально. Помимо формы `export default ...`, экспортирующей выражение со значением привязки, все остальные экспортируют привязки к локальным идентификаторам. В этом случае изменение значения переменной внутри модуля после экспорта приводит к тому, что внешняя импортированная привязка обращается к обновленному значению:

```
var foo = 42;
export { foo as default };

export var bar = "hello world";

foo = 10;
bar = "cool";
```

При импорте этого модуля результаты экспорта функций `default` и `bar` будут привязаны к локальным переменным `foo` и `bar`, то есть покажут обновленные значения 10 и "cool" независимо от того, какими они были на момент экспорта и на момент импорта. Привязки — это живые ссылки, потому важно только их значение на момент обращения к ним.



Двусторонняя привязка недопустима. Если после импорта из модуля переменной `foo` вы попытаетесь поменять значение импортированной переменной, появится сообщение об ошибке. Мы еще поговорим об этом в следующем разделе.

Можно повторно экспортировать элементы, экспортированные из другого модуля, например:

```
export { foo, bar } from "baz";  
export { foo as F00, bar as BAR } from "baz";  
export * from "baz";
```

Эти формы — аналог импорта из модуля "baz" и составления списка членов для экспорта из вашего модуля. Однако в приведенном примере члены модуля "baz" никогда не будут импортированы в локальную область видимости вашего модуля; они как бы пройдут насквозь без каких-либо изменений.

Импорт членов API

Для импорта модулей очевидным образом используется ключевое слово `import`. Этот процесс, как и экспорт, имеет несколько вариаций, так что не поленитесь как следует изучить следующий материал и поэкспериментировать с собственным кодом.

Для импорта конкретных именованных членов API модуля в область видимости верхнего уровня применяется следующий синтаксис:

```
import { foo, bar, baz } from "foo";
```



Синтаксис `{ .. }` в данном случае напоминает синтаксис объектного литерала или даже деструктуризации объекта. Но эта форма специально предназначена для модулей, не надо путать ее с другими вариантами фигурных скобок.

Строка "foo" называется *спецификатором модуля* (module specifier). Так как нам нужен статически анализируемый синтаксис, роль спецификатора играет строковый литерал; здесь нельзя использовать переменную, содержащую строковое значение.

С точки зрения кода ES6 и движка JS содержимое строкового литерала непрозрачно и бессмысленно. Загрузчик модуля интерпрети-

тирует эту строку как инструкцию по поиску модуля, содержащую или адрес URL, или маршрут к файлу в локальной файловой системе.

Перечисленные здесь идентификаторы `foo`, `bar` и `baz` должны совпадать с элементами именованного экспорта API модуля (применяется статический анализ и утверждение ошибки). В текущей области видимости они связаны как идентификаторы верхнего уровня:

```
import { foo } from "foo";

foo();
```

Вы можете переименовать импортированные связанные идентификаторы:

```
import { foo as theFooFunc } from "foo";

theFooFunc();
```

Если модуль обладает только результатами экспорта по умолчанию, которые вы хотите импортировать и привязать к идентификатору, можно по вашему выбору опустить `{ .. }` для этого связывания. Тогда синтаксис примет самую приятную и краткую форму:

```
import foo from "foo";

// или:
import { default as foo } from "foo";
```



Как объяснялось в предыдущем разделе, ключевое слово `default` в процедуре экспорта модуля задает именованный экспорт, в котором фигурирует имя `default`, как это показано во втором, более многословном варианте. Изменение имени с `default` на другое (в данном случае на `foo`) в явном виде выполняется в последнем фрагменте и неявно в первом.

Кроме того, при наличии у модуля соответствующего определения можно импортировать результаты экспорта по умолчанию вместе с результатами именованного экспорта. Вспомним это уже встречавшееся нам определение модуля:

```
export default function foo() { .. }  
  
export function bar() { .. }  
export function baz() { .. }
```

Давайте импортируем результаты экспорта по умолчанию этого модуля и два результата именованного экспорта.

```
import FOOFN, { bar, baz as BAZ } from "foo";  
  
FOOFN();  
bar();  
BAZ();
```

Крайне рекомендуется придерживаться подхода из философии модулей в ES6, который гласит, что импортировать следует только конкретные привязки. Если модуль предоставляет 10 методов API, а вам требуются только два из них, считается, что перетаскивать весь набор привязок API — пустая трата ресурсов.

Такой ограниченный импорт не только делает код более явным, но и увеличивает надежность статического анализа и распознавания ошибок (например, случайного использования некорректного имени привязки). Разумеется, это всего лишь рекомендация, никто не заставляет вас ее придерживаться.

Многие разработчики скажут, что такой подход — более трудоемкий, требующий регулярного пересмотра и обновления операторов импорта каждый раз, когда вам требуется еще какой-нибудь элемент модуля. Такова обратная сторона удобства.

В свете сказанного выше более предпочтительным может оказаться вариант, когда вы импортируете все содержимое модуля в одно

пространство имен вместо импорта отдельных членов в область видимости. К счастью, у оператора `import` есть синтаксическая вариация, поддерживающая такой стиль работы с модулем. Она называется *импортом пространства имен* (namespace import).

Допустим, модуль "foo" экспортируется следующим образом:

```
export function bar() { .. }  
export var x = 42;  
export function baz() { .. }
```

Вы можете импортировать API целиком в единую привязку к пространству имен модуля:

```
import * as foo from "foo";  
  
foo.bar();  
foo.x; // 42  
foo.baz();
```



Оператор `* as ..` требует группового символа `*`. Другими словами, нельзя, к примеру, написать `import { bar, x } as foo from "foo"` для извлечения некоторой части API, оставаясь привязанным к пространству имен `foo`. Я хотел бы, чтобы подобная возможность была, но в ES6 импорт в пространство имен происходит по принципу «всё или ничего».

Если модуль, который импортируется с помощью выражения `* as ..`, обладает результатом экспорта по умолчанию, этот результат в указанном пространстве имен называется **default**. Можно дополнительно именовать импорт по умолчанию извне привязки пространства имен как идентификатор верхнего уровня. Рассмотрим модуль "world", экспортированный следующим образом:

```
export default function foo() { .. }  
export function bar() { .. }  
export function baz() { .. }
```

А это процедура импорта:

```
import foofn, * as hello from "world";
foofn();
hello.default();
hello.bar();
hello.baz();
```

Это корректный синтаксис, но ситуацию запутывает тот факт, что один из методов модуля (экспорт по умолчанию) привязан к верхнему уровню области видимости, в то время как остальные результаты именованного экспорта (и один с именем `default`) привязаны как свойства к идентификатору пространства имен с другим названием (`hello`).

Я уже говорил, что лучше избегать такого способа проектирования экспорта модулей, чтобы пользователям не приходилось иметь дело с подобными странными вещами.

Все импортированные привязки неизменяемы и/или предназначены только для чтения. Рассмотрим результат предыдущего импорта (все последующие попытки присваивания будут приводить к появлению исключения `TypeError`):

```
import foofn, * as hello from "world";

foofn = 42;           // (runtime) TypeError!
hello.default = 42;   // (runtime) TypeError!
hello.bar = 42;       // (runtime) TypeError!
hello.baz = 42;       // (runtime) TypeError!
```

Выше, в разделе «Экспорт членов API», мы обсуждали, каким образом привязки `bar` и `baz` связаны с реальными идентификаторами внутри модуля `"world"`. Вы должны помнить, что если модуль изменяет эти значения, `hello.bar` и `hello.baz` начнут ссылаться на обновленные версии.

Локальные импортированные привязки неизменяемы и/или предназначены только для чтения, и при попытке что-нибудь с ними

сделать вы получаете ошибку `TypeError`. Это очень важно, так как без подобной защиты ваши изменения в конце концов повлияли бы на других пользователей модуля (напоминаю: он представляет собой синглтон), что потенциально могло бы привести к непредсказуемым результатам.

Более того, хотя модули в принципе могут менять свои члены API изнутри, к целенаправленному проектированию подобных модулей следует подходить с крайней осторожностью. Модули ES6 *задуманы* как статические, соответственно, все отклонения от этого принципа должны возникать как можно реже, и их следует тщательно и подробно документировать.



Существует философия проектирования модулей, согласно которой пользователям преднамеренно позволяют менять значения свойств API или же API модуля проектируется с возможностью «расширения» путем добавления в пространство имен API «подключаемых элементов». Как я уже сказал, API модулей ES6 следует рассматривать и проектировать как статические и неизменные, что сильно мешает введению альтернативных шаблонов проектирования модулей. Эти ограничения можно обойти, экспортировав обычный объект, а затем изменив его по своему желанию. Но будьте крайне осторожны и дважды подумайте, стоит ли идти по этой дороге.

Объявления, возникающие в результате импорта, считаются «приподнятыми» (см. книгу *Scope & Closures* этой серии). Рассмотрим пример:

```
foo();  
import { foo } from "foo";
```

Функцию `foo()` можно запускать, потому что статическое разрешение оператора `import ..` не только определило ее во время компиляции, но и «приподняло» объявление в верхнюю часть области видимости модуля, сделав эту функцию доступной во всем модуле.

Наконец, основная форма импорта выглядит следующим образом:

```
import "foo";
```

В данном случае привязки модуля не импортируются в вашу область видимости. Оператор загружает модуль `"foo"` (если тот еще не загружен), компилирует (если это еще не сделано) и оценивает его (если он еще не запускался).

В общем случае этот вид импорта не очень полезен. Возможны отдельные случаи, когда определение модуля сопровождается побочными эффектами (например, присваиванием какого-либо элемента окну или глобальному объекту). Кроме того, можно представить `import "foo"` как своего рода предварительную загрузку модуля, который понадобится вам в будущем.

Циклическая зависимость модулей

А импортирует В. В импортирует А. Как такое возможно?

Сразу скажу, что я стараюсь избегать проектирования систем с преднамеренной циклической зависимостью. Но я признаю, что это бывает оправданно и порой позволяет выйти из неприятных ситуаций.

Посмотрим, как с такими вещами справляется ES6. Начнем с модуля "А":

```
import bar from "B";

export default function foo(x) {
  if (x > 10) return bar( x - 1 );
  return x * 2;
}
```

А вот модуль "В":

```
import foo from "A";

export default function bar(y) {
  if (y > 5) return foo( y / 2 );
  return y * 3;
}
```

Если бы функции `foo(..)` и `bar(..)` находились в одной области видимости, мы имели бы дело со стандартной процедурой объявления функций. В данном случае объявления «приподняты» в общую область видимости и, соответственно, доступны друг для друга независимо от порядка разработки.

В случае модулей объявления находятся в разных областях видимости, и для обеспечения работоспособности циклических ссылок приходится прилагать дополнительные усилия.

Коротко говоря, циклические зависимости импорта проверяются и разрешаются следующим образом.

- Если первым загружается модуль "А", сначала файл сканируется и анализируется на предмет результатов экспорта, чтобы получить возможность зарегистрировать все доступные для импорта привязки. Затем обрабатывается выражение `import .. from "В"`, сообщающее, что пришло время извлечь модуль "В".
- После загрузки модуля "В" движок производит такой же анализ его привязок экспорта, как и в случае "А". Когда дело доходит до выражения `import .. from "А"`, API модуля "А" уже известен, поэтому остается удостовериться в корректности импорта. Когда же становится известен API модуля "В", появляется возможность проверить также и выражение `import .. from "В"` в ожидающем модуле "А".

По сути, взаимный импорт вкупе со статической проверкой обоих операторов импорта виртуально объединяет области видимости двух модулей (через привязки), так что функция `foo(..)` может вызывать функцию `bar(..)` и наоборот. Это сравнимо с ситуацией, когда оба модуля изначально объявляются в одной области видимости.

Теперь попробуем использовать два модуля вместе. Начнем с `foo(..)`:

```
import foo from "foo";  
foo( 25 ); // 11
```

Теперь попробуем `bar(..)`:

```
import bar from "bar";  
bar( 25 ); // 11.5
```

К моменту выполнения `foo(25)` или `bar(25)` анализ и компиляция всех модулей завершатся. Это означает, что функция `foo(..)` знает о функции `bar(..)`, а той известно о функции `foo(..)`.

Если нам требуется только взаимодействие с функцией `foo(..)`, достаточно будет импортировать модуль `"foo"`. Аналогичным образом обстоят дела с функцией `bar(..)` и модулем `"bar"`.

Разумеется, при желании мы *можем* импортировать обе функции:

```
import foo from "foo";  
import bar from "bar";  
  
foo( 25 ); // 11  
bar( 25 ); // 11.5
```

Семантика статической загрузки оператора `import` означает, что зависящие друг от друга модули `"foo"` и `"bar"` в результате импорта будут гарантированно загружены, проанализированы и скомпилированы до запуска любого из них. То есть циклическая зависимость разрешается статически, и все работает ожидаемым образом.

Загрузка модуля

В начале раздела «Модули» я уже писал, что оператор `import` использует предоставляемый средой размещения (браузером, Node.js и т. п.) механизм для превращения строки спецификатора в инструкцию по поиску и загрузке нужного модуля. Этот механизм называется системным *загрузчиком модуля* (module loader).

Загрузчик, по умолчанию предоставляемый средой в браузере, будет интерпретировать спецификатор модуля как URL-адрес, а тот, что на сервере, например Node.js (в общем случае), — как

локальный путь в файловой системе. Поведение по умолчанию предполагает, что загружаемый файл разработан в формате стандартных модулей ES6.

Кроме того, загружать модули в браузер можно при помощи тега HTML, аналогичного тому, который в настоящее время загружает программы сценариев. На момент написания книги не совсем понятно, будет ли это тег `<scripttype="module">` или тег `<module>`. Решение не имеет отношения к ES6, его обсуждение ведется соответствующим комитетом по стандартам.

Независимо от того, какой тег в итоге выберут, он будет использовать загрузчик по умолчанию (или индивидуально настроенный, который вы предварительно укажете).

Как и теги, используемые в разметке, загрузчик модуля не описан в ES6. Для этого есть отдельный стандарт (см. <http://whatwg.github.io/loader/>), в настоящее время контролируемый группой WHATWG, устанавливающей стандарты браузеров.

То, о чем вы прочитаете ниже, на момент написания книги было первым вариантом проектирования API и вполне может поменяться в дальнейшем.

Загрузка модулей извне

Один из вариантов непосредственного взаимодействия с загрузчиком модулей — это ситуация, когда нечто, не являющееся модулем, должно загрузить модуль. Рассмотрим пример:

```
// обычный сценарий в браузере загружается через '<script>',  
// оператор 'import' здесь недопустим
```

```
Reflect.Loader.import( "foo" ) // возвращает обещание для '"foo"'  
.then( function(foo){  
    foo.bar();  
} );
```


Служебная программа `Reflect.Loader.import(..)` целиком импортирует модуль в именованный параметр (как пространство имен), аналогично тому, как рассмотренное ранее выражение `import * as foo ..` осуществляло импорт в пространство имен.



Служебная программа `Reflect.Loader.import(..)` возвращает обещание, которое считается выполненным, когда модуль готов к работе. Для импорта набора модулей можно объединить обещания от нескольких вызовов `Reflect.Loader.import(..)` с помощью метода `Promise.all([..])`. Подробно обещания будут рассматриваться в главе 4.

Служебную программу `Reflect.Loader.import(..)` допускается использовать в реальном модуле, чтобы выполнить динамическую/условную загрузку там, где оператор `import` не работает. Например, можно выбрать для загрузки модуль, содержащий полизаполнение для функциональной особенности из ES7+, если ее тестирование покажет, что она не определяется текущим движком.

По соображениям производительности желательно избегать динамической загрузки, так как она затрудняет способность движка JS запускать раннюю выборку из своего статического анализа.

Специализированная загрузка

Другой случай прямого взаимодействия с загрузчиком модуля — ситуация, когда вам нужно настроить его поведение посредством изменения конфигурации или даже переопределения.

На момент написания книги был разработан полизаполнитель для API загрузчика модуля (см. <https://github.com/ModuleLoader/es6-module-loader>). Его характеристики пока немногочисленны и, скорее всего, со временем поменяются, но мы посмотрим, на что можно в итоге рассчитывать.

Вызов служебной программы `Reflect.Loader.import(...)` допускает второй аргумент, задающий различные варианты настройки задач импорта/загрузки. Например:

```
Reflect.Loader.import( "foo", { address: "/path/to/foo.js" } )  
.then( function(foo){  
    // ..  
} )
```

Также ожидается, что возможность настройки будет предоставлена (с помощью каких-то средств) для подключения к процессу загрузки модуля, в котором после загрузки, но до момента компиляции есть вероятность трансляции/транскомпиляции.

Например, можно загрузить модуль, формат которого пока не совместим с ES6 (например, CoffeeScript, TypeScript, CommonJS, AMD). На этапе трансляции он будет преобразован в ES6-совместимый модуль для дальнейшей обработки движком.

Классы

Практически с момента возникновения языка JavaScript его синтаксис и шаблоны разработки навевали мысли о возможной поддержке классов. Благодаря таким вещам, как операторы `new` и `instanceof`, а также свойству `.constructor`, возникает соблазн считать, что в JS существуют классы, скрытые где-то внутри системы прототипов.

Разумеется, «классы» JS не имеют ничего общего с обычными классами. Все различия хорошо документированы, поэтому я не буду тратить времени на их описание.



Чтобы лучше изучить шаблоны, с помощью которых в JS имитируются классы, и получить альтернативное представление о прототипах и о том, что называется делегированием, читайте вторую половину книги *this & Object Prototypes* этой серии.

Ключевое слово **class**

Хотя механизм прототипов в JS не умеет работать как традиционные классы, это не уменьшает количество просьб расширить синтаксическое удобство до такой степени, чтобы представление о «классах» больше совпадало с реальностью. Давайте рассмотрим появившееся в ES6 ключевое слово **class** и связанный с ним механизм.

Эта функциональная особенность появилась после бурных и продолжительных дебатов и стала компромиссом между несколькими противоположными точками зрения на подход к реализации классов в JS. Большинство разработчиков, мечтавших о полноценных классах, сочтут новый синтаксис весьма привлекательным, хотя и обнаружат отсутствие важных составляющих. Но поводов для волнения тут нет. Комитет TC39 уже работает над дополнительными функциональными особенностями, которые улучшат классы в следующих версиях стандарта.

В основе механизма классов, появившегося в ES6, лежит ключевое слово **class**, идентифицирующее блок, содержимое которого определяет члены прототипа функции. Рассмотрим пример:

```
class Foo {  
  constructor(a,b) {  
    this.x = a;  
    this.y = b;  
  }  
  
  gimmeXY() {  
    return this.x * this.y;  
  }  
}
```

Здесь нужно обратить внимание на следующие вещи.

- Запись **class Foo** влечет за собой создание (специальной) функции с именем **Foo**, во многом напоминающей те, что вы создавали до появления ES6.

- `constructor(...)` определяет сигнатуру функции `Foo(...)` и содержимое ее тела.
- Для методов класса применяется тот же самый синтаксис «кратких методов», что и для объектных литералов. Он обсуждался в главе 2. Сюда же входит краткая форма генератора, которую мы рассматривали выше, а также синтаксис методов чтения/записи из ES5. Но методы класса не являются перечисляемыми, в то время как методы объекта перечисляемы по умолчанию.
- В отличие от объектных литералов, в теле класса отсутствуют запятые, отделяющие члены друг от друга. Более того, запятые там вообще недопустимы.

Определение синтаксиса для ключевого слова `class` в предыдущем фрагменте можно грубо представить как существовавший до ES6 эквивалент, который, скорее всего, вполне знаком тем, кто пользовался в своем коде прототипами.

```
function Foo(a,b) {  
    this.x = a;  
    this.y = b;  
}  
  
Foo.prototype.gimmeXY = function() {  
    return this.x * this.y;  
}
```

Как в предшествующей ES6 форме, так и в новой «класс» допускает создание экземпляров, которые используются точно так, как следовало бы ожидать:

```
var f = new Foo( 5, 15 );  
  
f.x;           // 5  
f.y;           // 15  
f.gimmeXY();   // 75
```

Обратите внимание! Хотя класс `Foo` по виду сильно напоминает функцию `Foo()`, между ними есть ряд важных отличий.

- Обращение функции `Foo(..)` к классу `Foo` *должно* осуществляться через оператор `new`, так как существовавший до ES6 вариант `Foo.call(obj)` уже *не работает*.
- Если функция `Foo` допускает «поднятие» (см. книгу *Scope & Closures* этой серии), для класса `Foo` такая возможность отсутствует; оператор `extends ..` задает выражение, которое нельзя «поднять». Поэтому следует объявить класс, прежде чем создавать его экземпляры.
- `class Foo` наверху глобальной области видимости создает в ней лексический идентификатор `Foo`, но, в отличие от функции `Foo`, не создает свойства глобального объекта с таким именем.

Общепринятый оператор `instanceof` все еще работает с классами ES6, так как ключевое слово `class` всего лишь создает функцию-конструктор с таким же именем. Однако в ES6 появился способ настраивать работу оператора `instanceof` с помощью метода `Symbol.prototype.hasInstance` (см. раздел «Известные символы» главы 7).

Ключевое слово `class` можно рассматривать и другим способом, на мой взгляд, более удобным. Это своего рода *макрос*, который автоматически заполняет прототип объекта. По вашему желанию он также связывает соотношение `[[Prototype]]`, когда используется с ключевым словом `extends` (см. следующий раздел).

Класс ES6 не является реальной программной единицей, это метаконцепция, которая охватывает существующие программные единицы, такие как функции и свойства, и связывает их друг с другом.



Ключевое слово `class` может не только быть объявлением, но и находиться в составе выражения, например `var x = class Y { .. }`. В основном подобные вещи используются для передачи определения класса (с технической точки зрения самого конструктора) как аргумента функции или для присваивания его свойству объекта.

Ключевые слова `extends` и `super`

Классы ES6 имеют удобный синтаксис для установления делегирующей связи `[[Prototype]]` между двумя прототипами функций, часто неправильно называемой «наследованием» или (по непонятной причине) «наследованием прототипов». Для этого используется ориентированная на классы терминология, с которой вы уже знакомы, — ключевое слово `extends`:

```
class Bar extends Foo {
  constructor(a,b,c) {
    super( a, b );
    this.z = c;
  }

  gimmeXYZ() {
    return super.gimmeXY() * this.z;
  }
}

var b = new Bar( 5, 15, 25 );

b.x;           // 5
b.y;           // 15
b.z;           // 25
b.gimmeXYZ();  // 1875
```

А вот слово `super` — совершенно новое, оно позволяет делать вещи, которые до появления ES6 было невозможно реализовать напрямую (без кое-каких не слишком хороших и сложно достижимых компромиссов). В конструкторе это ключевое слово автоматически ссылается на «родительский конструктор». В предыдущем примере он назывался `Foo(..)`. Внутри метода `super` ссылается на «родительский объект», обеспечивая вам доступ к свойству/методу, например `super.gimmeXY()`.

Запись `Bar extends Foo`, разумеется, означает связывание `[[Prototype]]` свойства `Bar.prototype` со свойством `Foo.prototype`. Соответственно, ключевое слово `super` в таком методе, как `gimmeXYZ()`,

означает `Foo.prototype`, в то время как в конструкторе `Bar` оно указывает на `Foo`.



Применение ключевого слова `super` не ограничивается объявлениями классов. Работает оно и в объектных литералах, причем во многом тем же способом, который мы тут обсуждаем. Дополнительную информацию по этой теме вы найдете в разделе «Ключевое слово `super`» главы 2.

Super-драконы существуют

Нужно отметить, что поведение ключевого слова `super` зависит от того, в каком месте оно находится. К счастью, в большинстве случаев проблем с этим не возникает. Но в нетипичных ситуациях вы можете столкнуться в сюрпризами.

Бывает, что в конструкторе вам требуется ссылка на `Foo.prototype`, например, для прямого доступа к одному из его свойств/методов. Но в конструкторе ключевое слово `super` таким способом использовать нельзя; запись `super.prototype` не будет работать. Запись `super(..)` означает вызов новой функции `Foo(..)`, но ссылкой на саму функцию `Foo` это не является.

В качестве симметричного случая рассмотрим необходимость сослаться на функцию `Foo(..)` изнутри не принадлежащего конструктору метода. Написав `super.constructor`, мы укажем на функцию `Foo(..)`, но следует помнить, что вызываться она будет *только* оператором `new`. Запись `new super.constructor(..)` вполне корректна, но в большинстве случаев пользоваться ею нельзя, так как вы не можете заставить вызов ссылаться на являющийся его текущим контекстом объект или использовать его. А ведь именно это нам зачастую и требуется.

Кроме того, ключевое слово `super` выглядит как доступное для управления контекстом функции, подобно `this`, — то есть создается впечатление динамической связи между этими двумя случаями.

Но, в отличие от ключевого слова `this`, `super` динамическим не является. Когда конструктор или метод с его помощью создает внутри себя ссылку во время объявления (в теле класса), слово `super` статически связывается с иерархией конкретного класса и не допускает переопределения (по крайней мере, в ES6).

Что все это означает? Если у вас есть привычка брать метод одного «класса» и заимствовать для другого класса, переопределяя ключевое слово `this`, например, с помощью методов `call(..)` или `apply(..)`, то при наличии в заимствуемом методе ключевого слова `super` вы можете столкнуться с сюрпризами. Рассмотрим иерархию классов:

```
class ParentA {
  constructor() { this.id = "a"; }
  foo() { console.log( "ParentA:", this.id ); }
}

class ParentB {
  constructor() { this.id = "b"; }
  foo() { console.log( "ParentB:", this.id ); }
}

class ChildA extends ParentA {
  foo() {
    super.foo();
    console.log( "ChildA:", this.id );
  }
}

class ChildB extends ParentB {
  foo() {
    super.foo();
    console.log( "ChildB:", this.id );
  }
}

var a = new ChildA();
a.foo();           // ParentA: a
                   // ChildA: a
var b = new ChildB();
b.foo();           // ParentB: b
                   // ChildB: b
```


В этом фрагменте кода все кажется совершенно естественным. Но, если вы попытаетесь позаимствовать метод `b.foo()` и воспользоваться им в контексте `a` (в силу динамического связывания с помощью `this` подобное заимствование — достаточно распространенное явление, оно используется множеством способов, в том числе в такой вещи, как примеси), результат может вас удивить:

```
// заимствуем 'b.foo()', чтобы использовать в контексте 'a'  
b.foo.call( a );           // ParentB: a  
                           // ChildB: a
```

Как видите, ссылка `this.id` оказалась динамически изменена, в результате чего в обоих случаях было выведено `: a` вместо `: b`. Но ссылка `super.foo()` функции `b.foo()` динамически не поменялась, поэтому мы увидели `ParentB` вместо ожидаемого результата `ParentA`.

Так как `b.foo()` ссылается через ключевое слово `super`, оно статически привязывается к иерархии `ChildB/ParentB` и не может быть использовано на иерархическом уровне `ChildA/ParentA`. Обойти это ограничение средствами ES6 нельзя.

Действие ключевого слова `super` наглядно проявляется в случае статической иерархии классов без взаимного влияния. Но, по большому счету, такая гибкость — и есть основная причина использовать в коде ключевое слово `this`. Просто в случае комбинации `class` и `super` таких техник следует избегать.

Здесь нужно сузить процесс проектирования объектов до статических иерархий — ключевые слова `class`, `extends` и `super` в этом случае будут прекрасно работать. Еще один вариант — отказаться от имитации классов и использовать динамические гибкие бесклассовые объекты и делегирование `[[Prototype]]` (см. книгу *this & Object Prototypes* данной серии).

Конструктор подкласса

Для классов и подклассов конструкторы не требуются; если конструктор по умолчанию опустить, в обоих случаях он будет заме-

щен. Однако вид замещенного конструктора по умолчанию зависит от того, с каким классом — непосредственным или расширенным — мы имеем дело.

В частности, используемый по умолчанию конструктор подкласса автоматически вызывает родительский конструктор и передает любые аргументы туда. Другими словами, этот конструктор можно представить примерно вот так:

```
constructor(...args) {  
    super(...args);  
}
```

Отметим важную деталь. Не во всех языках программирования, поддерживающих классы, конструктор подклассов автоматически вызывает родительский конструктор. В C++ подобное имеет место, а вот в Java — нет. Еще важнее тот факт, что в версиях классов, существовавших до ES6, подробного тоже не было. Так что будьте осторожны с преобразованиями классов в стандарт ES6, если в вашем коде такого автоматического вызова не предполагается.

В ES6 на конструкторы подклассов накладывается еще одно странное ограничение. Доступ к ключевому слову `this` в таком конструкторе появляется только после вызова метода `super(..)`. Это поведение имеет крайне сложные причины, но все главным образом упирается в тот факт, что именно родительский конструктор создает и инициализирует значение `this` для вашего экземпляра. До ES6 это работало в обратную сторону: объект `this` создавался «конструктором подклассов», а затем вы вызвали родительский конструктор в контексте `this` этого подкласса.

Для иллюстрации рассмотрим пример. Так работали до ES6:

```
function Foo() {  
    this.a = 1;  
}  
  
function Bar() {  
    this.b = 2;  
    Foo.call( this );  
}
```

```
}  
// 'Bar' "расширяет" 'Foo'  
Bar.prototype = Object.create( Foo.prototype );
```

А вот такое в ES6 недопустимо:

```
class Foo {  
    constructor() { this.a = 1; }  
}  
  
class Bar extends Foo {  
    constructor() {  
        this.b = 2; // недопустимо до вызова 'super()'  
        super(); // для исправления нужно поменять местами эти  
                  два оператора  
    }  
}
```

В данном случае исправить ситуацию несложно. Достаточно поменять местами два оператора в конструкторе подкласса `Bar`. Но если в варианте кода, который имел место до появления ES6, вы полагались на возможность пропустить вызов «родительского конструктора», будьте осторожны, так как теперь ее нет.

Расширение встроенных объектов

Одним из самых многообещающих преимуществ в новом варианте использования ключевых слов `class` и `extend` стала долгожданная возможность превращать в подклассы встроенные объекты, например `Array`. Рассмотрим пример:

```
class MyCoolArray extends Array {  
    first() { return this[0]; }  
    last() { return this[this.length - 1]; }  
}  
  
var a = new MyCoolArray( 1, 2, 3 );  
  
a.length;    // 3  
a;           // [1,2,3]  
  
a.first();   // 1  
a.last();    // 3
```

До ES6 имитация «подкласса» `Array` путем создания объекта вручную и связывания со свойством `Array.prototype` работала лишь частично. Не удавалось воспроизвести особые поведения настоящих массивов, например автоматическое обновление свойства `length`. Имейте в виду, подклассы ES6 должны полноценно работать с унаследованными и новыми поведением.

Другое распространенное ограничение «подклассов» до ES6 было связано с объектом `Error`. Оно состояло в создании пользовательских «подклассов» ошибок. В момент появления настоящие объекты `Error` автоматически захватывают из стека специальную информацию, в том числе о номере строки и о файле, где появилась ошибка. У возникавших до ES6 пользовательских «подклассов» ошибок такого поведения не наблюдалось, что несколько ограничивало их применимость.

Тут нам поможет ES6:

```
class Oops extends Error {  
  constructor(reason) {  
    this.oops = reason;  
  }  
}  
  
// позднее:  
var ouch = new Oops( "I messed up!" );  
throw ouch;
```

Пользовательский объект `ouch` в приведенном фрагменте будет вести себя как настоящий объект ошибки, в том числе и захватывать информацию из стека. Это существенное улучшение!

Свойство `new.target`

В ES6 появилась новая концепция, называемая *метасвойством* (meta property). Она имеет форму `new.target` и подробно будет рассматриваться в главе 7.

Добавление к ключевому слову точки `.`, да и само имя свойства выглядят странно для JS.

Свойство `new.target` представляет собой новое «магическое» значение, доступное во всех функциях, хотя в обычных функциях оно всегда равняется `undefined`. В любом конструкторе `new.target` всегда будет указывать на конструктор, непосредственно вызвавший оператор `new`, даже если тот располагается в параллельном классе и был делегирован через вызов `super(..)` из дочернего конструктора.

Рассмотрим пример:

```
class Foo {
  constructor() {
    console.log( "Foo: ", new.target.name );
  }
}

class Bar extends Foo {
  constructor() {
    super();
    console.log( "Bar: ", new.target.name );
  }
  baz() {
    console.log( "baz: ", new.target );
  }
}

var a = new Foo();
// Foo: Foo

var b = new Bar();
// Foo: Bar <-- учитывает сторону, вызвавшую 'new'
// Bar: Bar

b.baz();
// baz: undefined
```

Метасвойство `new.target` в конструкторах классов не имеет особого назначения, кроме обеспечения доступа к свойству/методу `static` (см. следующий раздел).

Если `new.target` равняется `undefined`, значит, функция с помощью оператора `new` не вызывалась. После этого при необходимости можно принудительно организовать вызов `new`.

Ключевое слово `static`

Мы уже видели, что, когда подкласс `Bar` расширяет родительский класс `Foo`, объект `Bar.prototype` через `[[Prototype]]` связывается с объектом `Foo.prototype`. Кроме того, дополнительно функция `Bar()` связывается через `[[Prototype]]` с функцией `Foo()`. Второе утверждение может казаться не столь очевидным и логичным.

Тем не менее это бывает весьма полезно в случае, когда мы объявляем статические методы (не только свойства) для класса, так как они добавляются непосредственно в его объект-функцию, а не в объект-прототип этой функции. Рассмотрим пример:

```
class Foo {
  static cool() { console.log( "cool" ); }
  wow() { console.log( "wow" ); }
}
```

```
class Bar extends Foo {
  static awesome() {
    super.cool();
    console.log( "awesome" );
  }
  neat() {
    super.wow();
    console.log( "neat" );
  }
}
```

```
Foo.cool();           // "cool"
Bar.cool();           // "cool"
Bar.awesome();        // "cool"
                     // "awesome"
```

```
var b = new Bar();
```

```
b.neat();           // "wow"
                   // "neat"

b.awesome;          // undefined
b.cool;             // undefined
```

Следует помнить, что статические члены находятся в цепочке прототипа класса. По сути, они формируют параллельную цепочку между конструкторами функций.

Метод чтения конструктора в свойстве `Symbol.species`

Ключевое слово `static` может пригодиться нам при задании метода чтения в свойстве `Symbol.species` (в спецификации оно известно как `@@species`) для производного (дочернего) класса. Эта возможность позволяет дочернему классу передать в родительский информацию о том, каким конструктором следует пользоваться — когда вы не собираетесь задействовать конструктор самого дочернего класса, — если какой-либо метод родительского класса должен породить новый экземпляр.

Например, многие методы объекта `Array` создают и возвращают новые экземпляры `Array`. Если, определяя класс, производный от `Array`, вы хотите, чтобы эти методы продолжили создавать реальные экземпляры объекта `Array` вместо экземпляров производного класса, нужно сделать так:

```
class MyCoolArray extends Array {
  // принудительно превращаем 'species' в родительский
  // конструктор
  static get [Symbol.species]() { return Array; }
}

var a = new MyCoolArray( 1, 2, 3 ),
    b = a.map( function(v){ return v * 2; } );

b instanceof MyCoolArray; // false
b instanceof Array;       // true
```

Для иллюстрации того, каким образом метод родительского класса может использовать объявление конструктора дочернего компонента — примерно так, как это делает `Array#map(..)`, — рассмотрим пример:

```
class Foo {
  // передаем 'species' производному конструктору
  static get [Symbol.species]() { return this; }
  spawn() {
    return new this.constructor[Symbol.species]();
  }
}

class Bar extends Foo {
  // принудительно превращаем 'species' в родительский
  // конструктор
  static get [Symbol.species]() { return Foo; }
}

var a = new Foo();
var b = a.spawn();
b instanceof Foo;           // true

var x = new Bar();
var y = x.spawn();
y instanceof Bar;           // false
y instanceof Foo;           // true
```

Родительский класс `Symbol.species` не возвращает значение `this` для передачи в какой-либо производный класс, как можно было бы ожидать. Затем вручную переопределяется класс `Bar`, в результате чего для создания экземпляров начинает использоваться класс `Foo`. Разумеется, ничто не мешает производному классу создавать экземпляры самостоятельно при помощи оператора `new this.constructor(..)`.

Подводим итоги

В ES6 появился ряд функциональных особенностей, помогающих структуризации кода.

- Итераторы предоставляют последовательный доступ к данным или к операциям. Они могут использоваться новыми функциональными элементами, такими как цикл `for...of` и оператор `...`.
- Генераторы представляют собой функции, управляемые итератором и умеющие локально приостанавливать и возобновлять свою работу. Они позволяют программно (и интерактивно, через передачу сообщений `yield/next(...)`) генерировать значения, работа с которыми осуществляется посредством итераций.
- Модули дают возможность закрытой инкапсуляции деталей реализации при помощи открытых экспортируемых API. Определения модулей представляют собой экземпляры синглтонов на базе файлов, которые статически разрешаются во время компиляции.
- Классы предоставляют более чистый синтаксис для кода на базе прототипов. Добавление ключевого слова `super`, кроме того, разрешило сложную ситуацию с относительными ссылками в цепочке `[[Prototype]]`.

Эти новые инструменты — первое, на что вам следует обратить внимание, когда вы решите улучшить архитектуру ваших JS-проектов, перейдя к ES6.

4 Управление асинхронными операциями

Для тех, кто много пишет на языке JavaScript, не является секретом тот факт, что среди необходимых навыков фигурирует асинхронное программирование. Основным механизмом для управления асинхронностью раньше всегда был обратный вызов функции.

Но в ES6 появилась новая функциональная особенность, призванная устранить значительные недостатки, которыми обладает подход к асинхронности, основанный исключительно на обратных вызовах. Эта особенность называется *обещаниями* (promises). Кроме нее, ниже мы снова рассмотрим генераторы (они были подробно представлены в предыдущей главе) и шаблон их объединения с обещаниями, который позволяет сделать большой шаг вперед в программировании асинхронных операций на языке JavaScript.

Обещания

Для начала проясним кое-что. Обещания ориентированы не на замещение обратных вызовов. Это что-то вроде достойного доверия посредника для управления обратными вызовами — другими словами, они находятся между вызывающим кодом и выполняющим задачу асинхронным кодом.

Обещание можно представить и как слушателя, который позволяет вам подписаться на прослушивание события, дающего сигнал о завершении какой-либо задачи. Срабатывание возникает всего один раз, тем не менее его уместно рассматривать как событие.

Обещания можно соединять в цепочку, задающую последовательность асинхронно выполняемых задач. В комбинации с такими высокоуровневыми абстракциями, как метод `all(..)` (в классической терминологии «ворота») и метод `race(..)` (в классической терминологии «защелка»), цепочки обещаний предоставляют аппроксимацию управления асинхронными операциями.

Еще обещания можно представить как *будущее значение*, то есть обернутый вокруг значения независимый от времени контейнер. Этот контейнер рассматривается одним способом вне зависимости от того, окончательно или нет лежащее в его основе значение. Благодаря процессу наблюдения за обещанием это значение извлекается сразу же, как только становится доступным. Говоря другими словами, обещание представляет собой асинхронную версию значения, возвращаемого синхронной функцией.

Разрешение обещания завершается одним из двух способов: оно может быть выполнено или отклонено с необязательным единственным значением. Если обещание выполнено, окончательное значение называется осуществлением, в противном случае — причиной (как во фразе «причина для отказа»). Разрешение обещаний (осуществление или отказ) возможно только *один раз*. Дальнейшие попытки попросту будут проигнорированы. Соответственно, как только обещание оказывается разрешено, мы получаем недоступное для редактирования значение.

Очевидно, что существуют разные представления о том, что такое обещание. Но ни одно из них нельзя назвать исчерпывающим, каждое демонстрирует отдельный аспект целого. Основная идея состоит в том, что обещания значительно усовершенствовали обработку асинхронных операций, обеспечив порядок, предсказуемость и надежность.

Создание и использование обещаний

Для получения экземпляра обещания используется конструктор `Promise(..)`:

```
var p = new Promise( function(resolve,reject){
    // ..
} );
```

В конструкторе `Promise(..)` предусмотрены два параметра. Это функции, которые в общем случае называются `resolve(..)` и `reject(..)` соответственно. Они используются следующим образом.

- При вызове функции `reject(..)` обещание отклоняется, а переданное в нее значение становится причиной для отказа.
- Вызов функции `resolve(..)` без какого-либо значения или со значением, не являющимся обещанием, приводит к выполнению обещания.
- Если в вызываемую функцию `resolve(..)` передается еще одно обещание, основное обещание просто заимствует его состояние (исполнение или отказ) — вне зависимости от того, мгновенным или окончательным оно является.

Вот как обычно используются обещания для реорганизации обращения к функции, завязанного на обратный вызов. Начнем со служебной программы `ajax(..)`, которая обеспечивает обратный вызов, первым делом обрабатывающий ошибку:

```
function ajax(url,cb) {
    // делаем запрос, в конечном счете вызываем 'cb(..)'
}
// ..

ajax( "http://some.url.1", function handler(err,content){
    if (err) {
        // обрабатываем ошибку ajax
```

```
    }  
    else {  
        // обрабатываем удачное завершение 'contents'  
    }  
} );
```

Это можно преобразовать следующим образом:

```
function ajax(url) {  
    return new Promise( function pr(resolve,reject){  
        // делаем запрос, в конечном счете вызываем  
        // или 'resolve(..)', или 'reject(..)'  
    } );  
}  
  
// ..  
  
ajax( "http://some.url.1" )  
.then(  
    function fulfilled(contents){  
        // обрабатываем успешное завершение 'contents'  
    },  
    function rejected(reason){  
        // обрабатываем причину ошибки ajax  
    }  
);
```

Обещания обладают методом `then(..)`, принимающим одну или две функции обратного вызова. Первая (если она есть) интерпретируется как обработчик, который вызывается в случае успешного выполнения обещания, вторая (если она есть) — как обработчик, вызываемый в случае явного отказа или в ситуации, когда в процессе разрешения была перехвачена ошибка или исключение.

Если один из аргументов пропущен или не является допустимым значением функции — во втором случае обычно используется значение `null`, — по умолчанию будет вставлен эквивалентный заполнитель. Стандартный успешный обратный вызов передает свое значение завершения, а стандартный ошибочный обратный вызов — причину отказа. Сокращенный вид вызова `then(null,handleRejection)` выглядит так: `catch(handleRejection)`.

Методы `then(...)` и `catch(...)` автоматически конструируют и возвращают еще один экземпляр обещания, получающий результат разрешения, роль которого может играть как значение, возвращаемое после выполнения обещания, так и обработчик отказа (независимо от того, что на самом деле было вызвано). Рассмотрим пример:

```
ajax( "http://some.url.1" )
.then(
  function fulfilled(contents){
    return contents.toUpperCase();
  },
  function rejected(reason){
    return "DEFAULT VALUE";
  }
)
.then( function fulfilled(data){
  // обрабатываем данные от исходных обработчиков
  // обещания
} );
```

В этом фрагменте возвращается непосредственное значение метода `fulfilled(...)` или метода `rejected(...)`, которое на следующем шаге события получает метод `fulfilled(...)` второго метода `then(...)`. Если вместо этого вернуть новое обещание, оно будет классифицировано и принято как разрешение.

```
ajax( "http://some.url.1" )
.then(
  function fulfilled(contents){
    return ajax(
      "http://some.url.2?v=" + contents
    );
  },
  function rejected(reason){
    return ajax(
      "http://backup.url.3?err=" + reason
    );
  }
)
```

```
.then( function fulfilled(contents){  
    // 'contents' берется из следующего вызова  
    // 'ajax(..)', каким бы он ни был  
} );
```

Важно заметить, что исключение (или отклоненное обещание) в первом методе `fulfilled(..)` не приведет к вызову первого метода `rejected(..)`, так как данный обработчик отвечает только на разрешение в первом исходном обещании. Вместо этого отказ получит второе обещание, которое было вызвано на уровне второго метода `then(..)`.

В приведенном фрагменте не было слушателя для события отказа, это событие тихо сохранялось для дальнейших наблюдений. Если на него не обратить внимание, вызвав метод `then(..)` или `catch(..)`, оно останется необработанным. Некоторые консоли браузера могут распознавать такие необработанные отказы и сообщать о них, но опираться на это не следует; всегда нужно явно наблюдать за отказом выполнить обещание.



Это был краткий обзор теории и поведения обещаний. Для более глубокого исследования темы обратитесь к главе 3 книги *Async & Performance*.

Объекты `thenable`

Обещания — это истинные экземпляры конструктора `Promise(..)`. Но существуют еще и похожие на обещания объекты, которые называются *thenable* и в общем случае могут взаимодействовать с механизмом обещаний.

Термином *thenable* называется любой объект (или функция), обладающий методом `then(..)`. Везде, где механизмы обещаний могут допустить и принять состояние истинного обещания, они могут обрабатывать и объекты *thenable*.

По сути, этим термином называют любые значения, напоминающие обещания, но созданные не конструктором `Promise(..)`, а какой-то другой системой. С этой точки зрения в общем случае объекты `thenable` менее надежны, чем истинные обещания. В качестве примера давайте рассмотрим некорректно функционирующий объект `thenable`:

```
var th = {
  then: function thener( fulfilled ) {
    // бесконечно вызывает 'fulfilled(..)' каждые 100 мс
    setInterval( fulfilled, 100 );
  }
};
```

Получив этот объект `thenable` и соединив его в цепочку с методом `th.then(..)`, вы с удивлением обнаружите, что обработчик успешно завершённой операции вызывается в цикле, в то время как обычные обещания должны разрешаться всего один раз.

В общем случае, если вы получаете от какой-то системы некий элемент, который может оказаться обещанием или объектом `thenable`, не следует использовать его без проверки. В следующем разделе вы познакомитесь со служебной программой, появившейся в ES6 вместе с обещаниями, которая в подобных случаях помогает провести проверку.

Чтобы лучше понять опасность описанной ситуации, представьте, что *любой* объект в *произвольном* месте кода, обладающий методом `then(..)` или вызывающий его, потенциально может быть принят за `thenable`, если появляется вместе с обещаниями, даже если он вообще никак не связан с написанием асинхронного кода, использующего обещания.

До появления ES6 методы, вызывающие `then(..)`, в отдельную группу не выделялись. Понятно, что есть по меньшей мере несколько случаев, в которых имя метода было выбрано задолго до того, как возникла сама идея обещаний. С большой вероятностью за объекты `thenable` можно принять использующие метод `then(..)`

библиотеки асинхронного программирования, которые, строго говоря, не совместимы с обещаниями. Есть несколько таких библиотек.

Вы сами должны следить за тем, чтобы значения, некорректно принимаемые за объекты `thenable`, не использовались в механизме обещаний.

API обещаний

Существует API, предоставляющий статические методы для работы с обещаниями.

Метод `Promise.resolve(..)` возвращает объект, который выполнен с указанным значением. Давайте сравним принцип его работы с подходом, когда большая часть операций делается вручную.

```
var p1 = Promise.resolve( 42 );

var p2 = new Promise( function pr(resolve){
    resolve( 42 );
} );
```



Метод `Promise.resolve(..)` решает поднятую в предыдущем разделе проблему доверия. Любое значение, про которое вы не знаете наверняка, что оно является обещанием — даже если это непосредственное значение, — допускает нормализацию путем передачи в метод `Promise.resolve(..)`. Если значение распознается как обещание или объект `thenable`, его состояние/разрешение просто заимствуется, избавляя вас от проблем с некорректным поведением. Если же выясняется, что перед вами непосредственное значение, оно «оборачивается» в истинное обещание для обеспечения требуемого асинхронного поведения.

Объекты `p1` и `p2` будут обладать практически идентичным поведением. То же самое относится к разрешению с обещанием:

```
var theP = ajax( .. );

var p1 = Promise.resolve( theP );

var p2 = new Promise( function pr(resolve){
    resolve( theP );
} );
```

Метод `Promise.reject(..)` создает немедленно отклоненное обещание, так же, как его аналог, конструктор `Promise(..)`:

```
var p1 = Promise.reject( "Oops" );

var p2 = new Promise( function pr(resolve,reject){
    reject( "Oops" );
} );
```

Если методы `resolve(..)` и `Promise.resolve(..)` могут принять обещание и позаимствовать его состояние/разрешение, методы `reject(..)` и `Promise.reject(..)` не распознают, какое значение ими было получено. Поэтому при отказе в отношении обещания или объекта `thenable` в качестве причины указывается само обещание или объект `thenable` соответственно, а не его основное значение.

Метод `Promise.all([..])` принимает массив из одного или нескольких значений (непосредственных значений, обещаний или объектов `thenable`). Он возвращает обещание, которое будет выполнено при условии выполнения всех значений или отвергнуто, если отвергается хотя бы одно из них.

Рассмотрим эти значения/обещания:

```
var p1 = Promise.resolve( 42 );
var p2 = new Promise( function pr(resolve){
    setTimeout( function(){
        resolve( 43 );
    }, 100 );
} );
var v3 = 44;
var p4 = new Promise( function pr(resolve,reject){
    setTimeout( function(){
```

```
        reject( "Oops" );
    }, 10 );
} );
```

А теперь рассмотрим работу метода `Promise.all([..])` с комбинацией всех этих значений:

```
Promise.all( [p1,p2,v3] )
.then( function fulfilled(vals){
    console.log( vals );           // [42,43,44]
} );
```

```
Promise.all( [p1,p2,v3,p4] )
.then(
    function fulfilled(vals){
        // сюда мы никогда не попадаем
    },
    function rejected(reason){
        console.log( reason );     // Oops
    }
);
```

В то время как метод `Promise.all([..])` ожидает или принятия всех элементов, или первого отказа, метод `Promise.race([..])` ждет или первого принятия, или первого отказа. Рассмотрим пример:

*// ПРИМЕЧАНИЕ: повторно установим все тестовые значения,
// чтобы избежать отвлекающих проблем синхронизации!*

```
Promise.race( [p2,p1,v3] )
.then( function fulfilled(val){
    console.log( val );           // 42
} );
```

```
Promise.race( [p2,p4] )
.then(
    function fulfilled(val){
        // сюда мы никогда не попадаем
    },
    function rejected(reason){
        console.log( reason );     // Oops
    }
);
```



Если метод `Promise.all([])` будет выполнен сразу же (без каких-либо значений), то метод `Promise.race([])` заикнется. Это странное противоречие наводит на мысль, что данные методы в принципе нельзя использовать с пустыми массивами.

Генераторы и обещания

Для управления асинхронными операциями в программе наборы обещаний можно соединить в цепочку. Рассмотрим пример:

```
step1()
  .then(
    step2,
    step2Failed
  )
  .then(
    function(msg) {
      return Promise.all( [
        step3a( msg ),
        step3b( msg ),
        step3c( msg )
      ] )
    }
  )
  .then(step4);
```

Впрочем, существует и лучший вариант управления потоком асинхронных операций, причем с точки зрения стиля написания кода он предпочтительнее длинных цепочек обещаний. Для этого потребуются генераторы, с которыми мы познакомились в главе 3.

Нужно понять важный шаблон: генератор способен сформировать обещание, после чего можно сделать так, чтобы завершающее значение обещания возобновляло работу генератора.

Давайте посмотрим, как управление асинхронными операциями из предыдущего фрагмента кода осуществляется с помощью генератора:

```
function *main() {  
  var ret = yield step1();  
  
  try {  
    ret = yield step2( ret );  
  }  
  catch (err) {  
    ret = yield step2Failed( err );  
  }  
  
  ret = yield Promise.all( [  
    step3a( ret ),  
    step3b( ret ),  
    step3c( ret )  
  ] );  
  
  yield step4( ret );  
}
```

На первый взгляд этот код более многословен, чем эквивалентная цепочка обещаний из ранее приведенного фрагмента. Но в данном случае предлагается куда более привлекательный (и, что важнее, более понятный и обоснованный), выглядящий синхронным стиль написания кода — с операторами присваивания = «возвращаемых» значений и т. п. Особенно это заметно в том, что обработка ошибок блоком `try..catch` может использоваться по ту сторону скрытых границ асинхронности.

Почему мы используем обещания вместе с генератором, ведь написать асинхронный код для генератора можно и без них?

Обещания представляют собой надежную систему, которая переворачивает инверсию управления обычных обратных вызовов или преобразователей (см. книгу *Async & Performance* этой серии). Вот почему объединение надежности обещаний с синхронностью кода генераторов позволяет эффективно избавиться от всех основных недостатков обратных вызовов. Кроме того, такие служебные программы, как `Promise.all([..])`, позволяют красиво реализовать параллелизм на одном шаге `yield` генератора.

Как же все это работает? Нам требуется средство запуска, которое активирует наш генератор, получит выданное обещание и сделает так, чтобы оно или возобновляло работу генератора значением, полученным при успешном выполнении, или создавало внутри генератора ошибку с причиной отказа.

Подобное средство запуска существует во многих служебных программах и библиотеках, умеющих работать с асинхронным кодом; например, это функция `Q.spawn(..)` и подключаемое расширение `runner(..)`.

Однако сейчас мы рассмотрим отдельное средство запуска, чтобы проиллюстрировать весь процесс:

```
function run(gen) {
  var args = [].slice.call( arguments, 1), it;

  it = gen.apply( this, args );

  return Promise.resolve()
    .then( function handleNext(value){
      var next = it.next( value );

      return (function handleResult(next){
        if (next.done) {
          return next.value;
        }
        else {
          return Promise.resolve( next.value )
            .then(
              handleNext,
              function handleErr(err) {
                return Promise.resolve(
                  it.throw( err )
                )
                  .then( handleResult );
              }
            )
        }
      })( next );
    } );
}
```



Более подробно данная служебная программа рассматривается в книге *Async & Performance* этой серии. Кроме того, ее запуск с различными асинхронными библиотеками дает куда более впечатляющие результаты, чем в нашем примере. Скажем, `runner(...)` может обрабатывать обещания, последовательности, преобразователи и непосредственные (не являющиеся обещаниями) значения, обеспечивая потрясающую гибкость.

Словом, теперь запустить функцию `*main()` из предыдущего фрагмента кода совсем легко:

```
run( main )
.then(
  function fulfilled(){
    // '*main()' успешно завершена
  },
  function rejected(reason){
    // Ой, что-то пошло не так
  }
);
```

По сути, везде, где есть два и более асинхронных шага в потоке управляющей логики, можно и *нужно* использовать для управления в синхронном стиле выдающий обещания генератор, приводимый в действие запускающей программой. Это сильно облегчит понимание и поддержку кода.

Шаблон «выдача обещаний — возобновление работы генератора» однажды станет настолько распространенным и мощным инструментом, что в следующей версии языка JavaScript почти наверняка появится новый тип функции, реализующий все это автоматически, без сторонних запускающих программ. Асинхронные функции (скорее всего, именно так их назовут) мы рассмотрим в главе 8.

Подводим итоги

Так как язык JavaScript продолжает развиваться и расти, все чаще и чаще возникает потребность в асинхронном программировании.

Для решения такого рода задач зачастую недостаточно обратных вызовов, в сложных случаях они попросту не справляются.

К счастью, в ES6 появились обещания, позволяющие устранить один из основных недостатков обратных вызовов: невозможность полагаться на предсказуемое поведение. Обещания представляют собой будущее значение завершения потенциально асинхронной задачи и нормализуют поведение на границе синхронности и асинхронности.

Впрочем, только комбинация обещаний с генераторами в полной мере претворила в жизнь выгоды от перестройки кода управления асинхронным потоком, ослабив и абстрагировав ситуацию с глубоко вложенными обратными вызовами (так называемый «callback hell»).

В настоящее время управление этими взаимодействиями осуществляется с помощью различных средств запуска из асинхронных библиотек, но в конечном счете JavaScript начнет поддерживать такие шаблоны исключительно средствами синтаксиса.

5 Коллекции

Структурированный сбор данных и доступ к ним — это важный компонент практически любой программы JS. С начала существования языка и до сегодняшнего дня основными механизмами для создания структур данных были массив и объект. Разумеется, поверх строились многочисленные высокоуровневые структуры, такие как библиотеки пространства пользователя.

В ES6 в виде собственных компонентов языка были добавлены некоторые наиболее полезные (и оптимизирующие производительность) абстрактные представления структур данных.

Эту главу я начну с рассмотрения класса `TypedArray`, который технически появился вместе с ES5 несколько лет назад, но был стандартизирован только как дополнение к библиотеке WebGL. Зато в ES6 эти вещи приняты непосредственно в спецификации языка, что дает им официальный статус.

Коллекции `map` напоминают объекты (пары «ключ/значение»), но на самом деле это всего лишь строка для ключа, а значение вы можете использовать любое — даже другой объект или коллекцию `map`. Коллекции `set` напоминают массивы (списки значений), но все значения в них уникальны; если вы добавляете дубликат, он просто игнорируется. Существуют и «слабые» (в плане сборки мусора) эквиваленты: `WeakMap` и `WeakSet`.

TypedArrays

Как написано в книге *Types & Grammar* этой серии, в JS есть набор встроенных типов, например `number` и `string`. При виде функциональной особенности с названием «типизированный массив» прежде всего возникает предположение, что перед нами массив значений указанного типа, например, состоящий исключительно из строк.

На самом деле типизированные массивы связаны с предоставлением структурированного доступа к бинарным данным с помощью напоминающей массив семантики (индексный доступ и т. п.). Слово «тип» в названии относится к «представлению», наложенному на тип набора битов, которое, по сути, представляет собой отображение того, как именно будут рассматриваться эти биты: как массив 8-битных целых со знаком, 16-битных целых со знаком и т. п.

Как сконструировать такую совокупность битов? Она называется буфером и чаще всего создается непосредственно конструктором `ArrayBuffer(...)`:

```
var buf = new ArrayBuffer( 32 );  
buf.byteLength; // 32
```

Здесь `buf` — бинарный буфер длиной в 32 байта (256 бит), который заранее инициализирован значениями 0. Единственный доступный способ взаимодействия с этим буфером — проверка его свойства `byteLength`.



Некоторые функциональные особенности веб-платформ используют или возвращают буферы массива, например `FileReader#readAsArrayBuffer(...)`, `XMLHttpRequest#send(...)` и `ImageData` (данные об элементе `canvas`).

Но в этот буфер массива можно сверху поместить «представление» в форме типизированного массива. Рассмотрим пример:

```
var arr = new Uint16Array( buf );  
arr.length; // 16
```

Здесь `arr` — это типизированный массив 16-битных целых без знака, отображенный на 256-битный буфер `buf`, то есть вы получаете 16 элементов.

Порядок байтов

Важно понимать, что типизированный массив `arr` отображается с учетом порядка байтов (от младшего к старшему или от старшего к младшему) платформы, на которой работает JS. Если бинарные данные созданы на платформе с одним, а интерпретируются на платформе с противоположным порядком байтов, это может стать проблемой.

Порядок определяется тем, где именно, справа или слева, в многобайтовом числе — например, в 16-битном целом без знака, которое мы создали в предыдущем фрагменте кода, — находится младший байт (набор из 8 бит).

К примеру, возьмем десятичное число 3085, для представления которого требуется 16 бит. При наличии всего одного 16-битного числового контейнера оно будет представлено в двоичной форме как 0000110000001101 (шестнадцатеричная форма 0c0d) вне зависимости от порядка байтов.

Но если представить 3085 в виде двух 8-битных чисел, порядок байтов сильно повлияет на способ сохранения в памяти:

- 0000110000001101 / 0c0d (от старшего к младшему);
- 0000110100001100 / 0d0c (от младшего к старшему).

Если вы получили представление числа 3085 в системе, где принят порядок от младшего к старшему, то есть 0000110100001100, а затем наложили поверх представление в системе с обратным порядком байтов, вы увидите значение 3340 (в случае основания 10) и 0d0c (в случае основания 16).

От младшего к старшему — самое распространенное представление в Интернете в наши дни, но есть и браузеры, в которых это не так. Важно, чтобы вы знали порядок байтов как у производителя, так и у потребителя фрагментов бинарных данных.

Вот быстрый способ от MDN, позволяющий проверить порядок байтов ваших сценариев JavaScript:

```
var littleEndian = (function() {  
    var buffer = new ArrayBuffer( 2 );  
    new DataView( buffer ).setInt16( 0, 256, true );  
    return new Int16Array( buffer )[0] === 256;  
})();
```

Функция `littleEndian` может принимать значение `true` или `false`; для большинства браузеров возвращается значение `true`. Этот тест использует объект `DataView(..)`, предлагающий более низкоуровневый, тщательный контроль над доступом (записью/чтением) к битам с точки зрения расположенного над буфером слоя. Третий параметр метода `setInt16(..)` дает объекту `DataView` информацию о том, какой порядок байтов ожидается для рассматриваемой операции.



Не путайте порядок байтов в буферах массивов на запоминающем устройстве с тем, как число представляется в программах на JS. Например, `(3085).toString(2)` возвращает значение «110000001101», которое с предполагаемыми четырьмя ведущими «0», по всей видимости, является представлением от старшего к младшему. На самом деле это представление основано на едином 16-битном представлении, а не на представлении в виде двух 8-битных байтов. Показанная выше проверка объекта `DataView` — самый лучший способ определения порядка байтов в среде JS.

Множественные представления

К одному буферу можно присоединить несколько представлений, например:

```
var buf = new ArrayBuffer( 2 );

var view8 = new Uint8Array( buf );
var view16 = new Uint16Array( buf );

view16[0] = 3085;
view8[0];           // 13
view8[1];           // 12

view8[0].toString( 16 );           // "d"
view8[1].toString( 16 );           // "c"

// меняем порядок (как в случае порядка байтов!)
var tmp = view8[0];
view8[0] = view8[1];
view8[1] = tmp;

view16[0];           // 3340
```

Конструкторы типизированных массивов имеют множество версий сигнатур. Раньше мы просто передавали их в существующий буфер. Но у этой формы есть и два дополнительных параметра: `byteOffset` и `length`. Другими словами, представление типизированного массива можно начать в месте, положение которого отлично от 0, и он вовсе не должен заполнять буфер целиком.

Эта техника полезна в случаях, когда буфер включает в себя бинарные данные с неравномерным размером или местоположением.

К примеру, рассмотрим бинарный буфер, в начале которого располагается 2-байтовое число (также известное как `word`), затем идут два числа по 1 байту, а завершает последовательность 32-битное число с плавающей точкой. Вот способ доступа к этим данным с различными представлениями, смещениями и длинами, находящимся в одном буфере:

```
var    first = new Uint16Array( buf, 0, 2 )[0],
      second = new Uint8Array( buf, 2, 1 )[0],
      third = new Uint8Array( buf, 3, 1 )[0],
      fourth = new Float32Array( buf, 4, 4 )[0];
```

Конструкторы типизированных массивов

Кроме показанной в предыдущем разделе формы (`buffer`, [`offset`, [`length`]]) конструкторы типизированных массивов могут выглядеть следующим образом:

- `[constructor\](length)` — создает новое представление в новом буфере с числом байт `length`;
- `[constructor\](typedArr)` — создает новое представление и новый буфер и копирует содержимое из представления `typedArr`;
- `[constructor\](obj)` — создает новое представление и буфер и в цикле просматривает напоминающий массив объект `obj` с целью копирования его содержимого.

На момент появления ES6 доступны следующие конструкторы типизированных массивов:

- `Int8Array` (8-битные целые со знаком), `Uint8Array` (8-битные целые без знака);
– `Uint8ClampedArray` (8-битные целые без знака, со значениями в диапазоне 0-255);
- `Int16Array` (16-битные целые со знаком), `Uint16Array` (16-битные целые без знака);
- `Int32Array` (32-битные целые со знаком), `Uint32Array` (32-битные целые без знака);
- `Float32Array` (32-битные с плавающей точкой, IEEE-754);
- `Float64Array` (64-битные с плавающей точкой, IEEE-754).

Экземпляры конструкторов типизированных массивов практически совпадают с обычными встроенными массивами. Отличие состоит в фиксированной длине и значениях одного и того же типа.

Однако те и другие пользуются практически одинаковыми методами прототипов. Это означает, что с большой вероятностью вы

сможете иметь дело с типизированными массивами как с обычными, не прибегая к преобразованию.

Например:

```
var a = new Int32Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

a.map( function(v){
    console.log( v );
} );
// 10 20 30

a.join( "-" );
// "10-20-30"
```



Некоторые методы `Array.prototype` нельзя использовать с объектами `TypedArray`, например модификаторы (`splice(..)`, `push(..)` и т. п.) или `concat(..)`.

Имейте в виду, что элементы типизированных массивов и в самом деле ограничены объявленным размером битов. Если элементу массива `Uint8Array` вы пытаетесь присвоить значение, размер которого превышает 8 бит, значение будет обернуто таким образом, чтобы остаться в указанных пределах длины.

Это может стать проблемой, например, при операции возведения в квадрат всех значений типизированного массива. Рассмотрим пример:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = a.map( function(v){
    return v * v;
} );

b;
// [100, 144, 132]
```

У значений 20 и 30 после возведения в квадрат появится разряд переполнения. Обойти это ограничение позволяет функция `TypedArray#from(..)`:

```
var a = new Uint8Array( 3 );
a[0] = 10;
a[1] = 20;
a[2] = 30;

var b = Uint16Array.from( a, function(v){
    return v * v;
} );

// [100, 400, 900]
```

Более подробно метод `Array.from(..)`, используемый вместе с типизированными массивами, будет рассматриваться в разделе «Статическая функция `Array.from(..)`» главы 6. При этом в разделе «Отображение» объясняется функция отображения, принимаемая в качестве второго аргумента.

Что интересно, объект `TypedArray` обладает методом `sort(..)` как обычные массивы, но по умолчанию тот выполняет сравнение с помощью численной сортировки, а не преобразует значения в строки для лексикографического сравнения. Например:

```
var a = [ 10, 1, 2, ];
a.sort(); // [1,10,2]

var b = new Uint8Array( [ 10, 1, 2 ] );
b.sort(); // [1,2,10]
```

Метод `TypedArray#sort(..)` принимает в качестве необязательного аргумента функцию сравнения, аналогично работающему таким же способом методу `Array#sort(..)`.

Карты

Если вы давно программируете на JS, то знаете, что объекты — это основной механизм для создания структур данных, представляющих собой неупорядоченные пары «ключ/значение», также из-

вестных как карты. При этом основной недостаток применения объектов в качестве карт — невозможность взять в качестве ключа нестроковое значение.

Рассмотрим пример:

```
var    m = {};  
  
var    x = { id: 1 },  
       y = { id: 2 };  
  
m[x] = "foo";  
m[y] = "bar";  
  
m[x];           // "bar"  
m[y];           // "bar"
```

Что здесь происходит? Оба объекта `x` и `y` превращены в строки `"[object Object]"`, так что в массиве `m` будет задан только один этот ключ.

Ранее имитация карт иногда реализовывалась поддержкой параллельного массива нестроковых значений наряду с массивом значений. Например:

```
var    keys = [], vals = [];  
  
var    x = { id: 1 },  
       y = { id: 2 };  
  
keys.push( x );  
vals.push( "foo" );  
  
keys.push( y );  
vals.push( "bar" );  
  
keys[0] === x;           // true  
vals[0];                 // "foo"  
  
keys[1] === y;           // true  
vals[1];                 // "bar"
```

Разумеется, вряд ли вы захотите управлять этими параллельными массивами вручную, значит, нужно определить структуру данных

с методами, которые будут осуществлять управление автоматически. Но кроме необходимости программировать все это лично, недостатком являлся еще и тот факт, что вместо постоянного времени доступа $O(1)$ мы получали линейное, то есть $O(n)$.

Теперь благодаря ES6 больше этого делать не требуется! Достаточно воспользоваться объектом `Map(..)`:

```
var m = new Map();

var x = { id: 1 },
    y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

m.get( x );           // "foo"
m.get( y );           // "bar"
```

Единственный недостаток — отсутствие возможности использовать синтаксис доступа в виде квадратных скобок `[]` для задания и извлечения значений. Но с этой работой удовлетворительно справляются методы `get(..)` и `set(..)`.

Для удаления элемента из карты применяется не оператор `delete`, а метод `delete(..)`.

```
m.set( x, "foo" );
m.set( y, "bar" );

m.delete( y );
```

Содержимое всей карты удаляется методом `clear()`. Для получения информации о длине карты (то есть о количестве ключей) используется свойство `size` (а не `length`).

```
m.set( x, "foo" );
m.set( y, "bar" );
m.size;           // 2

m.clear();
m.size;           // 0
```

Конструктор `Map(...)` также может получать итерируемый объект (см. раздел «Итераторы» главы 3), который должен сгенерировать список массивов, причем первый элемент каждого будет ключом, а второй — значением. Такой формат итераций идентичен генерируемому методом `entries()`, который мы рассмотрим в следующем разделе. В результате очень просто получить копию карты:

```
var m2 = new Map( m.entries() );

// то же самое, что и:
var m2 = new Map( m );
```

Так как экземпляр карты — итерируемый, а итератор, который используется в нем по умолчанию, такой же, как в методе `entries()`, более предпочтительна вторая, короткая форма.

Разумеется, в конструкторе `Map(...)` можно просто вручную задать список *элементов* (массив ключа / массивы значений):

```
var x = { id: 1 },
    y = { id: 2 };
var m = new Map( [
  [ x, "foo" ],
  [ y, "bar" ]
] );
m.get( x );           // "foo"
m.get( y );           // "bar"
```

Значения карт

Для получения списка значений карты применяется метод `values(...)`, возвращающий итератор. В главах 2 и 3 мы рассмотрели разные способы последовательной обработки итератора (напоминающей обработку массива), например оператор распределения `...` и цикл `for...of`. Кроме того, в разделе «Создание массивов и подтипы» главы 6 будет подробно рассмотрен метод `Array.from(...)`.

Рассмотрим пример.

```
var    m = new Map();
var    x = { id: 1 },
      y = { id: 2 };
m.set( x, "foo" );
m.set( y, "bar" );
var    vals = [ ...m.values() ];

vals;                                     // ["foo", "bar"]
Array.from( m.values() );                // ["foo", "bar"]
```

Как упоминалось в предыдущем разделе, элементы карты можно циклически просматривать с помощью метода `entries()` (или встроенного итератора карты). Например:

```
var    m = new Map();
var    x = { id: 1 },
      y = { id: 2 };
m.set( x, "foo" );
m.set( y, "bar" );
var    vals = [ ...m.entries() ];

vals[0][0] === x;                        // true
vals[0][1];                             // "foo"

vals[1][0] === y; // true
vals[1][1]; // "bar"
```

Ключи карт

Для получения списка ключей карты используется метод `keys()`, возвращающий итератор.

```
var    m = new Map();
var    x = { id: 1 },
```

```
y = { id: 2 };

m.set( x, "foo" );
m.set( y, "bar" );

var keys = [ ...m.keys() ];

keys[0] === x;           // true
keys[1] === y;           // true
```

Чтобы узнать, есть ли в карте конкретный ключ, используйте метод `has(..)`.

```
var    m = new Map();

var    x = { id: 1 },
       y = { id: 2 };

m.set( x, "foo" );

m.has( x );               // true
m.has( y );               // false
```

По сути, карты позволяют ассоциировать дополнительную информацию (значение) с объектом (ключом), не добавляя ее к самому объекту.

В качестве ключа карты может использоваться любое значение, но, как правило, это объекты, так как строки и другие примитивы уже зарезервированы в качестве ключей обычных объектов. Другими словами, вы, скорее всего, станете использовать для карт объекты, кроме случаев, когда некоторые или все ключи не должны быть объектами и поэтому больше подходят карты.



Если в качестве ключа карты используется объект, который позднее оказывается удаленным (исчезают все ссылки на него), для освобождения памяти при проходе сборщика мусора, карта все равно будет возвращать эту запись. Чтобы сделать запись карты доступной для сборщика мусора, потребуется ее удаление. В следующем разделе мы рассмотрим более подходящий на роль ключа объект `WeakMap`.

Объекты WeakMap

Существует «слабая» вариация карты — объект `WeakMap`, обладающий таким же внешним поведением, но отличающийся тем, как под него выделяется память (и, в частности, как работает механизм сборки мусора).

Слабые карты принимают в качестве ключей только объекты, и те удерживаются *слабо*: если такой объект удаляется сборщиком мусора, то удаляется и соответствующая запись в коллекции `WeakMap`. Но это не внешнее поведение, поскольку сборщик мусора подбирает только те объекты, на которые нет ни одной ссылки, — то есть вы не можете проверить, существует ли такой объект в коллекции `WeakMap`.

Во всем остальном API для слабых ссылок аналогичен, хотя и обладает дополнительными ограничениями.

```
var    m = new WeakMap();

var    x = { id: 1 },
       y = { id: 2 };

m.set( x, "foo" );

m.has( x );           // true
m.has( y );           // false
```

У коллекций `WeakMap` отсутствует свойство `size` и метод `clear()`, а еще нет итераторов для ключей, значений или элементов. Так что даже если вы сбросите ссылку `x`, что приведет к удалению соответствующей записи из коллекции `m` посредством сборщика мусора, определить, что именно происходит, будет невозможно. Вам останется только надеяться, что все пройдет нужным образом.

Объекты `WeakMap`, как и объекты `Map`, позволяют мягко связывать информацию с объектом. Но особенно полезны они бывают в случае с объектами, которые вы не можете полностью контролировать,

например с элементами DOM. Если объект, используемый вами в качестве ключа карты, допускает удаление и должен быть доступным для сборщика мусора, вам больше подойдет вариант `WeakMap`.

Важно заметить, что коллекция `WeakMap` слабо удерживает только ключи, но не значения. Рассмотрим пример:

```
var    m = new WeakMap();

var    x = { id: 1 },
        y = { id: 2 },
        z = { id: 3 },
        w = { id: 4 };

m.set( x, y );

x = null;           // { id: 1 } доступен для сборщика мусора
y = null;           // { id: 2 } доступен для сборщика мусора
                    // только потому, что { id: 1 }

m.set( z, w );

w = null;           // { id: 4 } недоступен для сборщика мусора
```

По этой причине, как мне кажется, таким коллекциям больше подошло бы название `WeakKeyMap`¹.

Объекты Set

Объекты `set` представляют собой коллекции уникальных значений (дубликаты здесь игнорируются).

API для них примерно такой же, как для карт. Вместо метода `set(...)` используется метод `add(...)` (что выглядит несколько забавно), а метод `get(...)` вообще отсутствует.

¹ Карта слабых ключей (англ.). — Примеч. ред.

Рассмотрим пример:

```
var    s = new Set();

var    x = { id: 1 },
       y = { id: 2 };

s.add( x );
s.add( y );
s.add( x );

s.size; // 2

s.delete( y );
s.size; // 1

s.clear();
s.size; // 0
```

Конструктор `Set(..)` напоминает конструктор `Map(..)`, так как может принимать итерируемый объект, например другой объект `set` или массив значений.

Но если конструктор `Map(..)` ожидает список элементов (массив ключа / массивы значений), то конструктору `Set(..)` нужен только список значений (массив значений):

```
var    x = { id: 1 },
       y = { id: 2 };

var    s = new Set( [x,y] );
```

Метод `get(..)` этой коллекции не требуется, так как элементы оттуда не извлекаются, а всего лишь проверяется наличие или отсутствие какого-либо из них с помощью метода `has(..)`:

```
var    s = new Set();

var    x = { id: 1 },
       y = { id: 2 };

s.add( x );

s.has( x ); // true
s.has( y ); // false
```




В методе `has(..)` используется почти такой же алгоритм сравнения, как в методе `Object.is(..)` (см. главу 6), но `-0` и `0` интерпретируются как одно значение, а не как разные.

Итераторы коллекций Set

Коллекции `set` обладают теми же методами итератора, что и карты. Поведение этих методов отличается, хотя его можно до некоторой степени назвать симметричным.

Рассмотрим пример:

```
var    s = new Set();

var    x = { id: 1 },
       y = { id: 2 };

s.add( x ).add( y );

var    keys = [ ...s.keys() ],
       vals = [ ...s.values() ],
       entries = [ ...s.entries() ];

keys[0] === x;
keys[1] === y;

vals[0] === x;
vals[1] === y;

entries[0][0] === x;
entries[0][1] === x;
entries[1][0] === y;
entries[1][1] === y;
```

В коллекциях `set` итераторы `keys()` и `values()` дают список уникальных значений. Итератор `entries()` предоставляет список массивов записей, в котором оба элемента массива представляют собой уникальное значение коллекции. По умолчанию для коллекции `set` используется итератор `values()`.

Наиболее востребованное свойство этих коллекций — уникальность входящих в них элементов. Рассмотрим пример:

```
var s = new Set( [1,2,3,4,"1",2,4,"5"] ),
    uniques = [ ...s ];

uniques;                                     // [1,2,3,4,"1","5"]
```

Из-за условия уникальности приведение типов здесь оказывается невозможным, и поэтому 1 и "1" рассматриваются как разные значения.

WeakSets

Если коллекции `WeakMap` слабо держат свои ключи (цепляясь при этом за значения), то коллекции `WeakSet` слабо держат значения (ключей у них просто нет).

```
var    s = new WeakSet();

var    x = { id: 1 },
        y = { id: 2 };

s.add( x );
s.add( y );

x = null;                                     // 'x' доступен для сборщика мусора
y = null;                                     // 'y' доступен для сборщика мусора
```

Значениями коллекции `WeakSet` могут быть только объекты, но ни в коем случае не примитивы, допустимые в коллекциях `set`.

Подводим итоги

В ES6 появился ряд полезных коллекций, обеспечивающих более эффективную и продуктивную работу со структурированными данными.

Типизированные массивы дают нам «представления» буферов бинарных данных, приведенные в соответствие с различными типами целых чисел, например 8-битные целые без знака и 32-битные с плавающей точкой. Доступ к массиву бинарных данных упрощает написание и поддержку операций, предоставляя несложный способ работы с такой информацией, как видео, аудио, данные холста и пр.

Карты представляют собой пары «ключ/значение», в которых роль ключа вместо строки/примитива может играть объект. Коллекции **set** — это уникальные списки значений (произвольного типа).

Объекты **WeakMap** — разновидность карт, где ключ (объект) удерживается слабо, так что сборщик мусора имеет возможность свободно удалить запись, если речь идет о последней ссылке на объект.

Соответственно, коллекции **WeakSet** — это коллекции **set**, в которых слабо удерживается значение, опять же для того, чтобы сборщик мусора мог без проблем удалить запись в случае последней ссылки на объект.

6 Дополнения к API

Взяв за основу преобразования значений в математических расчетах, стандарт ES6 добавил к различным встроенным элементам и объектам множество статических свойств и методов, помогающих в решении распространенных задач. Кроме того, экземпляры некоторых встроенных объектов получили новые возможности через новые методы у их прототипов.



Большинство таких функциональных особенностей допускают успешное полизаполнение. Детально мы эту тему здесь рассматривать не будем, вы можете поискать совместимые со стандартами библиотеки Shims здесь: <https://github.com/paulmillr/es6-shim/>.

Массив

Тип `Array` в JS чаще всего расширяется посредством различных пользовательских библиотек. Неудивительно, что в ES6 к нему был добавлен ряд вспомогательных функциональных особенностей, как статических, так и прототипов (экземпляров).

Статическая функция `Array.of(..)`

У конструктора `Array(..)` есть всем известная странность. Если передать ему только один числовой аргумент, будет создан не массив из одного элемента, а пустой массив со свойством `length`, равным переданному числу. Это приводит к возникновению проблемного поведения «пустых слотов», которое так ругают, говоря о массивах в JS.

Поэтому функция `Array(..)` теперь заменена функцией `Array.of(..)`, где нет особенности, связанной с единственным числовым аргументом. Рассмотрим пример:

```
var a = Array( 3 );
a.length;           // 3
a[0];               // undefined

var b = Array.of( 3 );
b.length;           // 1
b[0];               // 3

var c = Array.of( 1, 2, 3 );
c.length;           // 3
c;                  // [1,2,3]
```

При каких обстоятельствах нам может потребоваться метод `Array.of(..)` вместо обычного создания массива при помощи литерального синтаксиса, например `c = [1,2,3]`? Есть два возможных случая.

Во-первых, это наличие обратного вызова, который должен охватывать переданный в массив аргумент или аргументы. Метод `Array.of(..)` идеально подходит под такие условия. Это не очень распространенная ситуация, но порой возникает именно такая необходимость.

Второй случай — наличие производного класса `Array` (см. раздел «Классы» в главе 3), в экземпляре которого вы хотите создавать и инициализировать элементы. Например:

```
class MyCoolArray extends Array {
  sum() {
    return this.reduce( function reducer(acc,curr){
      return acc + curr;
    }, 0 );
  }
}

var x = new MyCoolArray( 3 );
x.length;           // 3 - ой!
x.sum();             // 0 - ой!

var y = [3];         // Array, не MyCoolArray
y.length;           // 1
y.sum();             // 'sum' - это не функция

var z = MyCoolArray.of( 3 );
z.length;           // 1
z.sum();             // 3
```

Невозможно простым способом получить для класса `MyCoolArray` конструктор, переопределяющий поведение родительского конструктора `Array`, поскольку тот должен на самом деле создавать значения массива с регулярным поведением (инициализируя ключевое слово `this`). Прекрасным решением в этом случае оказывается «унаследованный» статический метод `of(..)` производного класса `MyCoolArray`.

Статическая функция `Array.from(..)`

Напоминающим массив объектом в JavaScript называется объект, обладающий свойством `length`, значения которого представлены целым числом от нуля и выше.

Работать с подобными структурами в JS всегда было неудобно; в общем случае требовалось сделать преобразование в обычный массив для доступа к методам прототипа `Array.prototype` (`map(..)`, `indexOf(..)` и т. п.). Обычно этот процесс выглядел примерно так:

```
// объект, напоминающий массив
var   arrLike = {
    length: 3,
    0: "foo",
    1: "bar"
};

var arr = Array.prototype.slice.call( arrLike );
```

Еще одна распространенная задача, в которой часто используется метод `slice(..)`, — дублирование настоящего массива:

```
var arr2 = arr.slice();
```

В обоих случаях более понятным и элегантным, а также более лаконичным решением послужит появившийся в ES6 метод `Array.from(..)`:

```
var arr = Array.from( arrLike );
var arrCopy = Array.from( arr );
```

Метод `Array.from(..)` проверяет, является ли первый аргумент итерируемым (см. раздел «Итераторы» в главе 3). Если это так, итератор используется для генерации значений, которые «копируются» в возвращаемый массив. Реальные массивы имеют для таких случаев встроенный итератор, применяющийся автоматически.

Но если передать в качестве первого аргумента методу `Array.from(..)` объект, напоминающий массив, он поведет себя как метод `slice()` (не имеющий аргументов) или метод `apply(..)`, то есть просто начнет перебирать значения в цикле, обращаясь к именованным свойствам, имена которых начинаются с 0 и заканчиваются значением свойства `length`.

Рассмотрим пример:

```
var   arrLike = {
    length: 4,
    2: "foo"
};
```

```
Array.from( arrLike );  
// [ undefined, undefined, "foo", undefined ]
```

Так как позиций 0, 1 и 3 в объекте `arrLike` не существует, для каждого из этих слотов результатом становится значение `undefined`.

Аналогичный результат можно получить следующим образом:

```
var emptySlotsArr = [];  
emptySlotsArr.length = 4;  
emptySlotsArr[2] = "foo";  
  
Array.from( emptySlotsArr );  
// [ undefined, undefined, "foo", undefined ]
```

Как избежать пустых слотов

В последнем фрагменте кода можно заметить небольшую, но крайне существенную разницу между массивом `emptySlotsArr` и результатом вызова метода `Array.from(..)`. Метод `Array.from(..)` никогда не создает пустых слотов.

До ES6 создание массива, инициализированного до определенного слота значениями `undefined` (то есть без пустых слотов), требовало дополнительных действий:

```
var a = Array( 4 );  
// четыре пустых слота!  
  
var b = Array.apply( null, { length: 4 } );  
// четыре значения 'undefined'
```

Метод `Array.from(..)` упрощает эту задачу:

```
var c = Array.from( { length: 4 } );  
// четыре значения 'undefined'
```



Пустые слоты, как в последнем примере, работают с некоторыми функциями, хотя есть и функции, игнорирующие их (например, `map(..)`). Намеренно создавать пустые слоты не следует, так как это практически гарантированно приведет к непредсказуемому поведению вашего кода.

Отображение

Метод `Array.from(...)` умеет делать еще одну полезную вещь. Вторым аргументом, если он присутствует, представляет собой отображающий обратный вызов (почти то же самое, чего ожидает обычный метод `Array#map(...)`), который активируется, когда нужно отобразить/преобразовать каждое значение из источника в возвращаемый целевой объект. Рассмотрим пример:

```
var arrLike = {
  length: 4,
  2: "foo"
};

Array.from( arrLike, function mapper(val,idx){
  if (typeof val == "string") {
    return val.toUpperCase();
  }
  else {
    return idx;
  }
} );
// [ 0, 1, "FOO", 3 ]
```



Подобно другим методам массивов, принимающим обратные вызовы, `Array.from(...)` принимает необязательный третий аргумент, который, будучи заданным, определяет связывание через ключевое слово `this` для переданного в качестве второго аргумента обратного вызова. В противном случае этот аргумент принимает значение `undefined`.

Пример применения метода `Array.from(...)` для преобразования массива 8-битных в массив 16-битных значений вы найдете в разделе «TypedArrays» главы 5.

Создание массивов и подтипы

В последних разделах мы обсуждали методы `Array.of(...)` и `Array.from(...)`, которые по аналогии с конструктором создают новые

массивы. Но что они делают в подклассах? Что они создают: экземпляры базового класса `Array` или производный подкласс?

```
class MyCoolArray extends Array {
  ..
}
MyCoolArray.from( [1, 2] ) instanceof MyCoolArray;    // true

Array.from(
  MyCoolArray.from( [1, 2] )
) instanceof MyCoolArray;                             // false
```

Оба метода, `of(..)` и `from(..)`, используют для создания массива конструктор, из которого они были вызваны. Соответственно, если в качестве основы использовать метод `Array.of(..)`, получится экземпляр класса `Array`, в то время как метод `MyCoolArray.of(..)` даст вам экземпляр `MyCoolArray`.

В разделе «Классы» главы 3 рассматривалась настройка `@@species`, определенная для всех встроенных классов (например, для класса `Array`), которая в случае создания нового экземпляра используется всеми методами прототипов. Прекрасный пример — метод `slice(..)`:

```
var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;    // true
```

В общем случае, скорее всего, будет желательным поведение по умолчанию, но, как упоминалось в главе 3, при необходимости его можно переопределить.

```
class MyCoolArray extends Array {
  // принудительно превращаем 'species' в родительский
  // конструктор
  static get [Symbol.species]() { return Array; }
}

var x = new MyCoolArray( 1, 2, 3 );

x.slice( 1 ) instanceof MyCoolArray;    // false
x.slice( 1 ) instanceof Array;         // true
```

Важно отметить, что настройка `@@species` используется только для методов прототипов, таких как `slice(..)`. Методы `of(..)` и `from(..)` ее не применяют; они задействуют только связывание с помощью ключевого слова `this` (независимо от того, каким конструктором делается ссылка). Рассмотрим пример:

```
class MyCoolArray extends Array {  
    // принудительно превращаем 'species' в родительский  
    конструктор  
    static get [Symbol.species]() { return Array; }  
}  
  
var x = new MyCoolArray( 1, 2, 3 );  
  
MyCoolArray.from( x ) instanceof MyCoolArray;    // true  
MyCoolArray.of( [2, 3] ) instanceof MyCoolArray; // true
```

Метод прототипа `copyWithin(..)`

Новый модифицирующий метод `Array#copyWithin(..)` доступен для всех массивов (в том числе для типизированных, которые мы рассматривали в главе 5). Метод `copyWithin(..)` копирует фрагмент массива в другую позицию, переписывая бывшие там ранее значения.

Его аргументы — *target* (индекс позиции, в которую осуществляется копирование), *start* (начальный индекс позиции источника копируемых элементов) и по желанию *end* (конечный индекс позиции источника). В случае отрицательного значения любого элемента он начинает отсчитываться от конца массива.

Рассмотрим пример:

```
[1,2,3,4,5].copyWithin( 3, 0 );           // [1,2,3,1,2]  
[1,2,3,4,5].copyWithin( 3, 0, 1 );        // [1,2,3,1,5]  
[1,2,3,4,5].copyWithin( 0, -2 );          // [4,5,3,4,5]  
[1,2,3,4,5].copyWithin( 0, -2, -1 );      // [4,2,3,4,5]
```

Метод `copyWithin(..)` не увеличивает длину массива, как можно видеть в первом примере. При достижении конца массива копирование просто останавливается.

Вопреки распространенному мнению, копирование далеко не всегда выполняется слева направо (с возрастающим индексом). В случае перекрытия диапазонов источника элементов и цели возможно многократное копирование уже скопированных значений, что, очевидно, нельзя назвать желательным поведением.

Поэтому алгоритм избегает подобной ситуации путем копирования в обратном порядке. Рассмотрим пример:

```
[1,2,3,4,5].copyWithin( 2, 1 );           // ???
```

При смещении слева направо 2 будет скопировано, чтобы переписать 3, затем это скопированное значение 2 снова будет скопировано, чтобы переписать значение 4, затем таким же образом будет переписано значение 5, и в результате вы получите массив `[1,2,2,2,2]`.

Если же поменять направление работы алгоритма, то сначала будет скопировано 4, чтобы переписать значение 5, затем — 3, чтобы переписать 4, затем — 2, чтобы переписать 3, и в итоге мы получим `[1,2,2,3,4]`. Такой результат является более корректным с точки зрения наших ожиданий, но мы не сможем понять, как он был получен, ограничившись представлением о направлении копирования слева направо.

Метод прототипа `fill(..)`

Заполнение существующего массива полностью (или частично) одним значением поддерживается в ES6 при помощи метода `Array#fill(..)`:

```
var a = Array( 4 ).fill( undefined );  
a;  
// [undefined,undefined,undefined,undefined]
```

В качестве необязательных параметров метод `fill(..)` принимает значения начального и конечного индексов, определяющих заполняемый фрагмент массива:

```
var a = [ null, null, null, null ].fill( 42, 1, 3 );  
a;                                     // [null,42,42,null]
```

Метод прототипа `find(..)`

Самым распространенным способом поиска значения в массиве в общем случае был метод `indexOf(..)`, возвращающий индекс обнаруженного значения или `-1`, если оно в массиве отсутствует:

```
var a = [1,2,3,4,5];  
  
(a.indexOf( 3 ) !== -1);      // true  
(a.indexOf( 7 ) !== -1);      // false  
  
(a.indexOf( "2" ) !== -1);    // false
```

Поиск при помощи метода `indexOf(..)` требует строгого соответствия `===`, поэтому по значению `"2"` нельзя найти `2` и наоборот. Переопределить алгоритм поиска соответствия для метода `indexOf(..)` невозможно. Еще одна проблема заключается в необходимости вручную проверять равенство значению `-1`.



В книге *Types & Grammar* этой серии вы найдете описание интересной (и вызывающей противоречивые отзывы) техники обхода проблемы со значением `-1` с помощью оператора `~`.

С ES5 самым распространенным способом получения контроля над логической схемой поиска соответствий был метод `some(..)`. Он создает для каждого элемента обратный вызов функции, пока не получит в результате такого вызова значение `true` / истинное значение, после чего работа останавливается. Так как функцию

обратного вызова определяете вы сами, у вас есть полный контроль над способом сопоставлений.

```
var a = [1,2,3,4,5];

a.some( function matcher(v){
    return v == "2";
} );                                // true

a.some( function matcher(v){
    return v == 7;
} );                                // false
```

Но, к сожалению, в таком случае вы получаете только значение `true/false`, указывающее на обнаружение соответствия, однако само совпавшее значение не выводится.

Эту проблему решает появившийся в ES6 метод `find(..)`. В своей основе он функционирует аналогично методу `some(..)`, но после возвращения обратным вызовом значения `true` передается еще и значение массива.

```
var a = [1,2,3,4,5];

a.find( function matcher(v){
    return v == "2";
} );                                // 2

a.find( function matcher(v){
    return v == 7;
});                                // undefined
```

Кроме того, существует пользовательская функция `matcher(..)`, позволяющая сопоставлять такие сложные значения, как объекты.

```
var    points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
```

```
];

points.find( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );                                     // { x: 30, y: 40 }
```



Как и остальные методы массивов, принимающие обратные вызовы, `find(..)` обладает вторым необязательным аргументом, позволяющим определять связывание посредством `this` для переданного в качестве первого аргумента обратного вызова. В противном случае он получает значение `undefined`.

Метод прототипа `findIndex(..)`

В предыдущем разделе вы познакомились с методом `some(..)`, возвращающим `true/false` по результатам поиска в массиве, и с методом `find(..)`, возвращающим найденное при этом значение. Но зачастую нам требуется узнать еще и индекс найденного значения.

Такую задачу решает метод `indexOf(..)`, но управлять логической схемой поиска соответствий в данном случае мы не можем; в ней всегда используется точное равенство `===`. Поэтому в ES6 появился новый метод `findIndex(..)`:

```
var    points = [
    { x: 10, y: 20 },
    { x: 20, y: 30 },
    { x: 30, y: 40 },
    { x: 40, y: 50 },
    { x: 50, y: 60 }
];

points.findIndex( function matcher(point) {
    return (
        point.x % 3 == 0 &&
        point.y % 4 == 0
    );
} );
```

```
    });  
  } }); // 2  
  
points.findIndex( function matcher(point) {  
    return (  
      point.x % 6 == 0 &&  
      point.y % 7 == 0  
    );  
  } }); // -1
```

Для получения булева значения как результата поиска не требуется прибегать к условию `findIndex(..) != -1` (которое всегда применялось с методом `indexOf(..)`), ведь это значение уже предоставлено методом `some(..)`. Не требуется нам и запись `a[a.findIndex(..)]` для получения результата совпадения, поскольку с такой задачей справляется метод `find(..)`. Наконец, метод `indexOf(..)` следует применять в случае, когда нам требуется индекс строгого соответствия, в то время как метод `findIndex(..)` предоставляет нам индекс при более специализированных вариантах поиска.



Как и остальные методы массивов, принимающие обратные вызовы, метод `findIndex(..)` обладает вторым необязательным аргументом, позволяющим определять связывание посредством `this` для переданного в качестве первого аргумента обратного вызова. В противном случае он получает значение `undefined`.

Методы прототипа `entries()`, `values()`, `keys()`

В главе 3 вы увидели, как структуры данных с помощью итератора по определенному шаблону поэлементно перечисляют свои значения. Этот подход мы снова рассмотрели в главе 5, исследуя, как появившиеся в ES6 новые коллекции (`Map`, `Set` и т. п.) предоставляют методы для реализации различных видов итераций.

Так как объекты `Array` были стандартизированы еще до ES6, традиционно их не рассматривают в качестве коллекций, но они могут

называться таковыми благодаря наличию все тех же методов итератора: `entries()`, `values()` и `keys()`.

Рассмотрим пример:

```
var a = [1,2,3];

[...a.values()];           // [1,2,3]
[...a.keys()];             // [0,1,2]
[...a.entries()];          // [ [0,1], [1,2], [2,3] ]

[...a[Symbol.iterator]()]; // [1,2,3]
```

Как и в случае коллекции `Set`, в объектах `Array` по умолчанию используется тот же самый итератор, который возвращает метод `values()`.

Далее, в разделе «Функции проверки строки», вы увидите, что метод `Array.from(..)` рассматривает пустые слоты массива как имеющие значение `undefined`. Это связано со следующим способом работы итераторов массива:

```
var a = [];
a.length = 3;
a[1] = 2;

[...a.values()];           // [undefined,2,undefined]
[...a.keys()];             // [0,1,2]
[...a.entries()];          // [ [0,undefined], [1,2],
                             [2,undefined] ]
```

Объект

К классу `Object` были добавлены несколько статических вспомогательных функций. Традиционно считалось, что подобные функции ориентированы на поведения и характеристики значений объекта.

Но начиная с ES6 статические функции класса `Object` будут предназначаться для глобальных API общего назначения, которые не попадают естественным образом в какую-нибудь другую категорию (например, `Array.from(...)`).

Статическая функция `Object.is(..)`

Статическая функция `Object.is(..)` сравнивает значения еще более строгим образом, чем оператор `===`.

Метод `Object.is(..)` активизирует лежащий в его основе алгоритм `SameValue` (спецификация ES6, раздел 7.2.9). По сути он аналогичен алгоритму строгой операции сравнения `===` (спецификация ES6, раздел 7.2.13), но с двумя важными исключениями.

Рассмотрим пример:

```
var x = NaN, y = 0, z = -0;

x === x;           // false
y === z;           // true

Object.is( x, x ); // true
Object.is( y, z ); // false
```

Для строгих сравнений нужно пользоваться оператором `===`; не следует рассматривать метод `Object.is(..)` как его замену. Но в случаях, когда требуется строго идентифицировать значение `NaN` или `-0`, вам поможет `Object.is(..)`.



В ES6 также появился метод `Number.isNaN(..)` (он будет рассмотрен ниже), который в ряде случаев более удобен в качестве средства проверки; вы можете предпочесть вариант `Number.isNaN(x)` варианту `Object.is(x, NaN)`. Аккуратно проверить на значение `-0` позволяет громоздкая конструкция `x == 0 && 1 / x === -Infinity`, но лучше воспользоваться вариантом `Object.is(x, -0)`.

Статическая функция `Object.getOwnPropertySymbols(..)`

В разделе «Символы» главы 2 обсуждался появившийся в ES6 новый тип примитивных значений `Symbol`.

Скорее всего, символы будут использоваться в основном как специальные свойства (метасвойства) объектов. Поэтому в ES6 появился метод `Object.getOwnPropertySymbols(..)`, извлекающий из объектов исключительно их символьные свойства:

```
var    o = {
    foo: 42,
    [ Symbol( "bar" ) ]: "hello world",
    baz: true
};

Object.getOwnPropertySymbols( o );           // [ Symbol(bar) ]
```

Статическая функция `Object.setPrototypeOf(..)`

В главе 2 я также упоминал метод `Object.setPrototypeOf(..)`, который (вполне ожидаемо) задает прототип объекта `[[Prototype]]` для делегирования поведения (подробно это рассматривается в книге *this & Object Prototypes*). Рассмотрим пример:

```
var    o1 = {
    foo() { console.log( "foo" ); }
};
var    o2 = {
    // .. определение o2 ..
};

Object.setPrototypeOf( o2, o1 );

// делегирует в 'o1.foo()'
o2.foo();                               // foo
```

Альтернативный вариант:

```
var    o1 = {
    foo() { console.log( "foo" ); }
};

var    o2 = Object.setPrototypeOf( {
    // .. определение o2 ..
}, o1 );

// делегирует в 'o1.foo()'
o2.foo();                                // foo
```

В обоих примерах соотношение между `o2` и `o1` появляется в конце определения `o2`. Еще чаще соотношение между `o2` и `o1` задается в верхней части определения `o2`, как это происходит в случае классов и в специальной ссылке `__proto__` в объектном литерале (см. раздел «Установка `[[Prototype]]`» главы 2).



Как вы уже видели, разумно задавать `[[Prototype]]` сразу после создания объекта. А вот редактировать его впоследствии — не самая лучшая идея, она больше создает путаницу, чем проясняет ситуацию.

Статическая функция `Object.assign(..)`

Многие библиотеки и фреймворки JavaScript предоставляют различные средства копирования/смешивания свойств разных объектов (например, метод `extend(..)` объекта `jQuery`). Между этими средствами существуют тонкие различия, например, в отношении к свойству со значением `undefined`, которое может игнорироваться, а может и нет.

В ES6 появился метод `Object.assign(..)`, представляющий упрощенную версию этих алгоритмов. Его первый аргумент — целевой объект (*target*), а все остальные передаваемые аргументы — объекты-источники (*sources*), которые обрабатываются в порядке перечисления. Перечисляемые и собственные (то есть не «унаследованные») ключи каждого источника, в том числе и символы,

копируются так же, как и в случае оператора присваивания =. Метод `Object.assign(..)` возвращает целевой объект.

Рассмотрим пример настройки объекта:

```
var    target = {},
      o1 = { a: 1 }, o2 = { b: 2 },
      o3 = { c: 3 }, o4 = { d: 4 };

// устанавливаем свойство только для чтения
Object.defineProperty( o3, "e", {
  value: 5,
  enumerable: true,
  writable: false,
  configurable: false
} );

// устанавливаем неперечисляемое свойство
Object.defineProperty( o3, "f", {
  value: 6,
  enumerable: false
} );
o3[ Symbol( "g" ) ] = 7;

// устанавливаем неперечисляемый символ
Object.defineProperty( o3, Symbol( "h" ), {
  value: 8,
  enumerable: false
} );

Object.setPrototypeOf( o3, o4 );
```

В целевой объект будут скопированы только свойства a, b, c, e и `Symbol("g")`.

```
Object.assign( target, o1, o2, o3 );

target.a;           // 1
target.b;           // 2
target.c;           // 3

Object.getOwnPropertyDescriptor( target, "e" );
// { value: 5, writable: true, enumerable: true,
//   configurable: true }

Object.getOwnPropertySymbols( target );
// [Symbol("g")]
```

Свойства `d`, `f` и `Symbol("h")` при копировании опускаются; из операции присваивания исключаются все свойства, которые не являются перечисляемыми и собственными. Кроме того, `e` копируется как обычное свойство присваивания, а не как предназначенное только для чтения.

Выше я показывал, как с помощью метода `setPrototypeOf(..)` настроить соотношение вида `[[Prototype]]` между объектами `o2` и `o1`. Вот другой вариант установки этого взаимодействия, в котором используется метод `Object.assign(..)`:

```
var o1 = {
  foo() { console.log( "foo" ); }
};

var o2 = Object.assign(
  Object.create( o1 ),
  {
    // .. определение o2 ..
  }
);

// делегирует в 'o1.foo()'
o2.foo();                // foo
```



Метод `Object.create(..)` представляет собой стандартное средство ES5 для создания пустого объекта, связанного с `[[Prototype]]`. Более подробно он рассматривается в книге *this & Object Prototypes* этой серии.

Объект Math

В ES6 появилось несколько новых математических методов, заполняющих пробелы или помогающих в выполнении распространенных операций. Все эти вещи могут быть вычислены вручную, но теперь, когда для их расчета появились встроенные методы, в ряде случаев движок JS в состоянии или оптимизировать обсчет

результатов, или предоставить результат с большей десятичной точностью.

Скорее всего, эти методы главным образом будут востребованы кодом JS на подмножестве `asm.js` и транскомпилированным кодом (см. книгу *Async & Performance* этой серии), а не непосредственно разработчиками.

Тригонометрия:

`cosh(..)`

Гиперболический косинус числа.

`acosh(..)`

Гиперболический арккосинус.

`sinh(..)`

Гиперболический синус.

`asinh(..)`

Гиперболический арксинус.

`tanh(..)`

Гиперболический тангенс.

`atanh(..)`

Гиперболический арктангенс.

`hypot(..)`

Квадратный корень суммы квадратов (то есть обобщенный вид теоремы Пифагора).

Арифметика:

`cbrt(..)`

Кубический корень.

`clz32(..)`

Подсчет ведущих нулей в 32-битном бинарном представлении.

`expm1(..)`

То же самое, что и $\exp(x) - 1$.

`log2(..)`

Двоичный логарифм числа (с основанием 2).

`log10(..)`

Десятичный логарифм числа.

`log1p(..)`

То же самое, что и $\log(x + 1)$.

`imul(..)`

Произведение двух 32-битных целых чисел.

Мета-методы:

`sign(..)`

Возвращает знак числа.

`trunc(..)`

Возвращает целую часть числа.

`fround(..)`

Округляет до ближайшего 32-битного (с одинарной точностью) значения с плавающей точкой.

Объект Number

Для корректной работы ваших программ крайне важна точная обработка чисел. В ES6 появился ряд дополнительных свойств и функций для выполнения распространенных операций с числами.

Два дополнения к объекту `Number` — это обычные ссылки на ранее существовавшие глобальные функции `Number.parseInt(..)` и `Number.parseFloat(..)`.

Статические свойства

В ES6 в виде статических свойств были добавлены некоторые полезные численные константы:

`Number.EPSILON`

Минимальное значение между любыми двумя числами: 2^{-52} (в главе 2 книги *Types & Grammar* рассматривается применение этого значения в качестве допустимой погрешности в арифметических действиях с плавающей точкой).

`Number.MAX_SAFE_INTEGER`

Максимальное целое число, которое «безопасно» может быть представлено в JS: $2^{53} - 1$.

`Number.MIN_SAFE_INTEGER`

Минимальное целое число, которое «безопасно» может быть представлено в JS: $-(2^{53} - 1)$ или $(-2)^{53} + 1$.



Более подробно «безопасные» целые числа описываются в главе 2 книги *Types & Grammar* этой серии.

Статическая функция `Number.isNaN(..)`

Стандартная глобальная функция `isNaN(..)` некорректно работает с момента появления, она возвращает `true` не только для значения `NaN`, но и для не являющихся числами значений, так она осуществляет приведение аргумента к численному типу (что иногда дает в результате `NaN`). В ES6 появилась исправленная функция `Number.isNaN(..)`, работающая корректным образом.

```
var a = NaN, b = "NaN", c = 42;

isNaN( a );           // true
isNaN( b );           // true - ой!
isNaN( c );           // false

Number.isNaN( a );    // true
Number.isNaN( b );    // false - исправлено!
Number.isNaN( c );    // false
```

Статическая функция **Number.isFinite(..)**

Возникает соблазн воспринять название функции, как `isFinite(..)`, и предположить, что она просто «не бесконечна». Но на самом деле это не совсем верный взгляд. Появившаяся в ES6 новая функция куда сложнее. Рассмотрим пример:

```
var a = NaN, b = Infinity, c = 42;

Number.isFinite( a );    // false
Number.isFinite( b );    // false

Number.isFinite( c );    // true
```

Стандартная глобальная функция `isFinite(..)` выполняет приведение аргумента, в то время как `Number.isFinite(..)` таким поведением не обладает:

```
var a = "42";

isFinite( a );           // true
Number.isFinite( a );    // false
```

Бывают случаи, когда приведение необходимо, и тогда имеет смысл использовать функцию `isFinite(..)`. Альтернативный и, вероятно, более рациональный вариант — `Number.isFinite(+x)`, где переменная `x` в явном виде преобразуется в число до передачи в функцию (см. главу 4 книги *Types & Grammar* этой серии).

Статические функции, связанные с целыми числами

Числовые значения в JavaScript всегда имеют плавающую точку (стандарт IEE-754). Соответственно, концепция определения целого числа связана не с проверкой его типа, ведь JS таких различий попросту не делает.

Вместо этого нужно определить, существует ли отличная от нуля десятичная часть значения, что раньше обычно делалось следующим образом:

```
x === Math.floor( x );
```

В ES6 появилась вспомогательная функция `Number.isInteger(..)`, которая потенциально дает возможность решать этот вопрос с большей эффективностью:

```
Number.isInteger( 4 );           // true  
Number.isInteger( 4.2 );        // false
```



В JavaScript разницы между значениями 4, 4., 4.0 и 4.0000 нет. Все они будут рассматриваться как «целые» и давать значение `true` при проверке функцией `Number.isInteger(..)`.

Кроме того, функция `Number.isInteger(..)` отфильтровывает очевидно не являющиеся целыми значения, которые потенциально могут быть перепутаны в случае `x === Math.floor(x)`:

```
Number.isInteger( NaN );         // false  
Number.isInteger( Infinity );    // false
```

В ряде случаев работа с целыми числами важна, поскольку позволяет упростить некоторые виды алгоритмов. Сам по себе код JS, если в нем оставить только целые значения, быстрее работать не будет, но существуют техники оптимизации, доступные для движ-

ка (например, `asm.js`), в которых используются только значения типа `integer`.

Из-за обработки функцией `Number.isInteger(..)` значений `NaN` и `Infinity` определение функции `isFloat(..)` уже нельзя свести к `!Number.isInteger(..)`. Приходится делать, к примеру, следующие вещи:

```
function isFloat(x) {  
    return Number.isFinite( x ) && !Number.isInteger( x );  
}  
  
isFloat( 4.2 );           // true  
isFloat( 4 );             // false  
  
isFloat( NaN );           // false  
isFloat( Infinity );      // false
```



Это может показаться странным, но значение `Infinity` не следует рассматривать, ни как целое, ни как десятичное.

Кроме того, в ES6 появилась функция `Number.isSafeInteger(..)`, которая проверяет, является ли значение целым и при этом попадает ли оно в диапазон между `Number.MIN_SAFE_INTEGER` и `Number.MAX_SAFE_INTEGER` (включительно).

```
var    x = Math.pow( 2, 53 ),  
       y = Math.pow( -2, 53 );  
  
Number.isSafeInteger( x - 1 );    // true  
Number.isSafeInteger( y + 1 );    // true  
  
Number.isSafeInteger( x );        // false  
Number.isSafeInteger( y );        // false
```

Объект String

У объекта `String` и до ES6 было множество вспомогательных функций, а теперь их количество еще больше увеличилось.

Функции Unicode

В разделе «Операции со строками, поддерживающие Unicode» главы 2 детально обсуждались методы `String.fromCodePoint(..)`, `String#codePointAt(..)` и `String#normalize(..)`. Они были добавлены, чтобы улучшить поддержку Unicode в строковых значениях JS.

```
String.fromCodePoint( 0x1d49e );           // "𐀚"
"ab𐀚".codePointAt( 2 ).toString( 16 );     // "1d49e"
```

Метод прототипа строки `normalize(..)` используется для выполнения нормализации Unicode, которая или объединяет символы с соседними «комбинируемыми знаками», или выполняет их разделение.

В общем случае нормализация не оказывает эффекта на отображение строки, но меняет ее содержимое, что влияет на значение свойства `length` и на доступ к символу по его позиции.

```
var s1 = "e\u0301";
s1.length;           // 2

var s2 = s1.normalize();
s2.length;           // 1
s2 === "\xE9";       // true
```

Метод `normalize(..)` принимает необязательный аргумент, указывающий форму нормализации. Он может принимать одно из четырех значений: `"NFC"` (по умолчанию), `"NFD"`, `"NFKC"` или `"NFKD"`.



Обсуждение форм нормализации и эффекта, который они оказывают на строки, выходит за рамки этой книги. Дополнительные сведения по данной теме можно получить здесь: <http://www.unicode.org/reports/tr15/>.

Статическая функция `String.raw(..)`

Метод `String.raw(..)` предоставляется как встроенная теговая функция для использования с шаблонами строковых литералов (см. главу 2). Он позволяет получить необработанное строковое значение без какой-либо обработки escape-последовательностей.

Эта функция практически никогда не вызывается вручную, а используется с тегированными шаблонными строками.

```
var str = "bc";

String.raw`\ta${str}d\xE9`;
// "\tabcd\xE9", не " abcdé"
```

В итоговой строке мы получаем два отдельных необработанных символа `\` и `t`, вместо одного символа escape-последовательности `\t`. То же самое верно и для escape-последовательности Unicode.

Функция прототипа `repeat(..)`

В таких языках, как Python и Ruby, повторение строки реализуется следующим образом:

```
"foo" * 3;                                // "foofoofoo"
```

В JS этот подход не работает, так как символ умножения `*` определен только для чисел. Соответственно, в данном варианте кода строка `"foo"` будет приведена к значению `NaN`.

Впрочем, в ES6 появился метод прототипа строки `repeat(..)`, позволяющий решить эту задачу:

```
"foo".repeat( 3 );                        // "foofoofoo"
```

Функции проверки строки

В дополнение к существовавшим еще до ES6 методам `String#indexOf(..)` и `String#lastIndexOf(..)` появились три новых метода поиска/проверки: `startsWith(..)`, `endsWith(..)` и `includes(..)`.

```
var palindrome = "step on no pets";

palindrome.startsWith( "step on" );           // true
palindrome.startsWith( "on", 5 );             // true

palindrome.endsWith( "no pets" );            // true
palindrome.endsWith( "no", 10 );             // true

palindrome.includes( "on" );                 // true
palindrome.includes( "on", 6 );              // false
```

Для всех этих методов результат поиска пустой строки "" обнаруживается или в начале, или в конце строки.



По умолчанию эти методы не принимают в качестве строки поиска регулярные выражения. В разделе «Символы регулярных выражений» главы 7 вы найдете информацию об отключении проверки `isRegExp` в первом аргументе.

Подводим итоги

ES6 дал нам много дополнительных вспомогательных API-функций для различных встроенных объектов.

- Для массивов появились статические функции `of(..)` и `from(..)`, а также функции прототипов, например `copyWithin(..)` и `fill(..)`.
- К объектам были добавлены такие статические функции, как `is(..)` и `assign(..)`.
- Объект `Math` обогатился такими статическими функциями, как `acosh(..)` и `clz32(..)`.

- Объект `Number` получил как дополнительные статические свойства, например `Number.EPSILON`, так и статические функции, такие как `Number.isFinite(..)`.
- К объекту `String` были добавлены статические функции, например `String.fromCodePoint(..)` и `String.raw(..)`, а также функции прототипа, такие как `repeat(..)` и `includes(..)`.

Большинство этих новшеств допускает полизаполнения (см. библиотеку ES6 Shim) и появилось под влиянием служебных программ в распространенных библиотеках и фреймворках JS.

7 Метапрограммирование

Метапрограммированием называется программирование, при котором целью операций является поведение самой программы. Другими словами, это программирование программирования вашей программы. Громоздкое определение, не так ли?

Например, если вы проверяете соотношение между объектами *a* и *b* (связаны ли они через `[[Prototype]]`), используя выражение `a.isPrototype(b)`, это называется самоанализом и представляет собой вариант метапрограммирования. Еще один яркий пример — макросы (пока не существующие в JS), в которых код модифицирует сам себя во время компиляции. Нельзя не вспомнить и такие распространенные задачи метапрограммирования, как перечисление ключей объекта циклом `for...in` или проверку, является ли объект экземпляром «конструктора класса».

Метапрограммирование концентрируется на одной или нескольких задачах из следующего списка: код, анализирующий сам себя; код, модифицирующий сам себя; код, модифицирующий поведение языка по умолчанию и влияющий при этом на другой код.

Цель метапрограммирования — применение возможностей языка для того, чтобы сделать остальной код более осмысленным, выразительным и/или гибким. Из-за изменчивой природы этого про-

граммирования дать более точное определение не представляется возможным. Оно становится понятнее, если рассматривать примеры.

В ES6 к уже имеющимся функциональным особенностям JS были добавлены несколько новых форм/функций.

Имена функций

Бывают ситуации, когда при анализе кодом самого себя возникает вопрос, как называется некая функция. При этом ответ на него оказывается на удивление неоднозначным.

Рассмотрим пример:

```
function daz() {  
    // ..  
}  
  
var    obj = {  
    foo: function() {  
        // ..  
    },  
    bar: function baz() {  
        // ..  
    },  
    bam: daz,  
    zim() {  
        // ..  
    }  
};
```

Здесь ответ на вопрос о том, какое имя имеет функция `obj.foo()`, содержит ряд нюансов: это "foo", "" или `undefined`? А как насчет функции `obj.bar()`: она называется "bar" или "baz"? А функция `obj.bam()` носит имя "bam" или "daz"? А функция `obj.zim()`?

Более того, возникает вопрос, как обстоят дела с функциями, передаваемыми в качестве обратных вызовов, например:

```
function foo(cb) {  
    // как в данном случае называется 'cb()'?  
}  
  
foo( function(){  
    // Я анонимная!  
} );
```

Функции в программе бывают выражены различными способами, и четко и однозначно определить их имена не всегда возможно.

Куда важнее отличать, относится ли имя функции к свойству `name` — да, у функций есть такое свойство — или к имени привязки к лексической области видимости, как, к примеру, `bar` в выражении `function bar() { .. }`.

Имя привязки к лексической области видимости используется в таких операциях, как рекурсия.

```
function foo(i) {  
    if (i < 10) return foo( i * 2 );  
    return i;  
}
```

Для целей метапрограммирования применяется свойство `name`, поэтому в нашем обсуждении мы сфокусируемся именно на нем.

Путаница возникает сама собой, так как лексическое имя функции (если таковое имеется) одновременно задается как ее свойство `name`. При этом официального требования на подобное поведение в спецификации ES5 (и предшествующих ей) нет.

Задание свойства `name` было нестандартным, но весьма надежным средством. В ES6 его стандартизировали.



Если функции присвоено значение `name`, то именно оно обычно используется при трассировке стека в инструментах разработчика.

Логические выводы

Что происходит со свойством `name`, если у функции отсутствует лексическое имя?

В ES6 появились правила вывода, позволяющие определить корректное значение свойства `name` для присваивания даже при отсутствии у функции лексического имени.

Рассмотрим пример:

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

abc.name;                                // "abc"
```

Если бы функции было присвоено лексическое имя, например `abc = function def() { .. }`, свойство `name` имело бы значение `"def"`. Но при отсутствии лексического имени интуитивно корректным кажется имя `"abc"`.

А вот другие формы, которые будут (или нет) выводить имя в ES6:

```
(function(){ .. });                      // name:
(function*(){ .. });                     // name:
window.foo = function(){ .. };          // name:

class Awesome {
  constructor() { .. }                  // name: Awesome
  funny() { .. }                        // name: funny
}

var c = class Awesome { .. };           // name: Awesome

var o = {
  foo() { .. },                         // name: foo
  *bar() { .. },                        // name: bar
  baz: () => { .. },                     // name: baz
  bam: function(){ .. },                // name: bam
  get qux() { .. },                     // name: get qux
  set fuz() { .. },                     // name: set fuz
  ["b" + "iz"]:
```

```
    function(){ .. },           // name: biz
    [Symbol( "buz" )]:
    function(){ .. }           // name: [buz]
};

var    x = o.foo.bind( o );     // name: bound foo
(function(){ .. }).bind( o );   // name: bound

export default function() { .. } // name: default
var    y = new Function();      // name: anonymous
var    GeneratorFunction =
    function*({}).__proto__.constructor;
var    z = new GeneratorFunction(); // name: anonymous
```

Свойство `name` по умолчанию недоступно для записи, но вы можете выбрать его конфигурацию, воспользовавшись методом `Object.defineProperty(..)`.

Метасвойства

В разделе «Свойство `new.target`» главы 3 я рассказал про новую для JS концепцию, появившуюся в ES6: метасвойство. Как следует из названия, такие свойства предоставляют особую информацию, доступ к которой в противном случае был бы затруднен.

В случае свойства `new.target` ключевое слово `new` служит контекстом доступа к свойству. Очевидно, что само по себе слово `new` не является тем, что делает эту возможность особенной. Но при использовании свойства `new.target` внутри вызова конструктора (активации при помощи оператора `new` функции/метода) это ключевое слово превращается в виртуальный контекст, соответственно, свойство `new.target` может ссылаться на целевой конструктор, вызвавший оператор `new`.

Это наглядный пример метапрограммирования, так как здесь налицо намерение определить изнутри вызова конструктора, какой была исходная цель оператора `new`. Обычно это делается для самоанализа (исследования принадлежности к типу/структуре) или для доступа к статическому свойству.

К примеру, иногда требуется различное поведение в конструкторе в зависимости от того, как он вызывается — напрямую или через дочерний класс:

```
class Parent {
  constructor() {
    if (new.target === Parent) {
      console.log( "Создан экземпляр родителя" );
    }
    else {
      console.log( "Создан экземпляр потомка" );
    }
  }
}

class Child extends Parent {}

var a = new Parent();
// Создан экземпляр родителя

var b = new Child();
// Создан экземпляр потомка
```

В данном случае есть небольшой нюанс. Дело в том, что конструктор `constructor()` внутри определения класса `Parent` на самом деле получил лексическое имя класса (`Parent`), притом что с точки зрения синтаксиса класс — это отдельная от конструктора сущность.



Как всегда бывает в случаях метапрограммирования, лучше избегать написания слишком сложного кода, который может оказаться непонятным для вас по прошествии времени и/или для тех, кто займется его поддержкой. Описанные в этой главе приемы следует применять с осторожностью.

Известные символы

В разделе «Символы» главы 2 был рассмотрен появившийся в ES6 новый тип примитивов. В дополнение к символам, которые вы можете определять в своих программах, в JS появился и ряд встро-

енных символов, названных *известными символами* (WKS — well-known symbols).

Значения таких символов в основном предназначены для отображения метасвойств, предоставляющих дополнительный контроль над поведением кода в программах JS.

Далее мы коротко рассмотрим каждое свойство и обсудим его значение.

Symbol.iterator

В главах 2 и 3 я рассказывал о способах применения символа `@@iterator`, который автоматически используется оператором распространения `...` и циклом `for...of`. Кроме того, в главе 5 вы видели этот символ, определенный для новых коллекций, появившихся в ES6.

Символ `Symbol.iterator` представляет специальное положение (свойство) произвольного объекта, в котором алгоритмы языка будут автоматически искать метод, конструирующий экземпляр итератора, который работает со значениями объекта. У многих объектов он определен по умолчанию.

Кроме того, мы можем задать для любого значения объекта итератор с нашей собственной логической схемой, указав свойство `Symbol.iterator`, даже если при этом будет переопределен итератор, заданный по умолчанию. Смысл метапрограммирования состоит в том, что мы определяем поведение, которым другие части кода JS (а именно операторы и циклические конструкции) будут пользоваться, обрабатывая указанные нами значения объекта.

Рассмотрим пример:

```
var arr = [4,5,6,7,8,9];

for (var v of arr) {
  console.log( v );
}
```

```
// 4 5 6 7 8 9

// определяем итератор, продуцирующий значения
// только из нечетных индексов
arr[Symbol.iterator] = function*() {
    var idx = 1;
    do {
        yield this[idx];
    } while ((idx += 2) < this.length);
};

for (var v of arr) {
    console.log( v );
}
// 5 7 9
```

Symbol.toStringTag и Symbol.hasInstance

Одна из наиболее распространенных задач метапрограммирования — это анализ значений и определение их типа, позволяющий понять, какие операции с ними можно проделывать. Для анализа объектов чаще всего применяются метод `toString()` и оператор `instanceof`.

Рассмотрим пример:

```
function Foo() {}

var a = new Foo();

a.toString();           // [object Object]
a instanceof Foo;       // true
```

В ES6 появилась возможность управлять поведением этих операций:

```
function Foo(greeting) {
    this.greeting = greeting;
}

Foo.prototype[Symbol.toStringTag] = "Foo";
```



```
Object.defineProperty( Foo, Symbol.hasInstance, {
  value: function(inst) {
    return inst.greeting == "hello";
  }
} );

var    a = new Foo( "hello" ),
      b = new Foo( "world" );

b[Symbol.toStringTag] = "cool";

a.toString();           // [object Foo]
String( b );            // [object cool]

a instanceof Foo;       // true
b instanceof Foo;       // false
```

Символ `@@toStringTag` прототипа (или самого экземпляра) указывает строковое значение, которое будет использоваться в строковом описании объекта по умолчанию.

Символ `@@hasInstance` представляет собой метод в функции конструктора, который получает значение экземпляра объекта и возвращает `true` или `false`, тем самым показывая, можно ли считать его экземпляром.



Чтобы задать для функции символ `@@hasInstance`, используют метод `Object.defineProperty(..)`, так как по умолчанию у `Function.prototype` имеет место ситуация `writable: false`. Детально это объясняется в книге *this & Object Prototypes*.

Symbol.species

В разделе «Классы» главы 3 вы познакомились с символом `@@species`, контролирующим, каким конструктором воспользуются встроенные методы класса, которому нужно породить новые экземпляры.

Чаще всего этот символ требуется в ситуации, когда при наличии класса, производного от `Array`, нужно определить, какой конструк-

тор (`Array(...)` или подкласса) должен использовать такие унаследованные методы, как `slice(...)`. По умолчанию вызванный для экземпляра подкласса `Array` метод `slice(...)` породит новый экземпляр этого подкласса. Зачастую именно такое поведение и требуется.

Но можно прибегнуть к метaprogramмированию, переписав заданное по умолчанию определение символа `@@species` этого класса:

```
class Cool {
  // отдаем '@@species' производному конструктору
  static get [Symbol.species]() { return this; }

  again() {
    return new this.constructor[Symbol.species]();
  }
}

class Fun extends Cool {}

class Awesome extends Cool {
  // принудительно превращаем '@@species' в родительский
  // конструктор
  static get [Symbol.species]() { return Cool; }
}

var    a = new Fun(),
      b = new Awesome(),
      c = a.again(),
      d = b.again();

c instanceof Fun;           // true
d instanceof Awesome;       // false
d instanceof Cool;          // true
```

По умолчанию настройка `Symbol.species` задана для встроенных собственных конструкторов и обеспечивает поведение, проиллюстрированное выше во фрагменте определения класса `Cool`. В пользовательских классах она по умолчанию отсутствует, но, как вы убедились, ее поведение не так уж сложно эмулировать.

Для определения метода, генерирующего новые экземпляры, следует прибегнуть к метапрограммированию шаблона `new this.constructor[Symbol.species](...)`, а не к жесткой конструкции `new this.constructor(...)` или `new XYZ(...)`. После этого производные классы смогут настраивать символ `Symbol.species`, указывая, какой конструктор должен генерировать экземпляры.

Symbol.toPrimitive

В книге *Types & Grammar* этой серии обсуждается абстрактная операция приведения `ToPrimitive`, применяющаяся, когда объект нужно превратить в примитивное значение для дальнейшей обработки (например, сравнения `==` или сложения `+`). До появления ES6 средства контроля этого поведения отсутствовали.

В ES6 появился символ `@@toPrimitive`, который, как свойство любого объектного значения, можно настроить для приведения `ToPrimitive`, указав соответствующий метод.

Рассмотрим пример:

```
var arr = [1,2,3,4,5];

arr + 10;                                // 1,2,3,4,510

arr[Symbol.toPrimitive] = function(hint) {
  if (hint == "default" || hint == "number") {
    // суммируем все числа
    return this.reduce( function(acc,curr){
      return acc + curr;
    }, 0 );
  }
};

arr + 10;                                // 25
```

Метод `Symbol.toPrimitive` снабжен подсказками `"string"`, `"number"` или `"default"` (последнее должно интерпретироваться как `"number"`),

зависящими от того, операция какого типа ожидается при выполнении `ToPrimitive`. В последнем примере у операции сложения `+` подсказка отсутствовала (было передано значение `"default"`). Для операции умножения `*` следует использовать подсказку `"number"`, а для метода `String(arr)` — подсказку `"string"`.



Оператор `==` активирует для объекта при сравнении с необъектным значением операцию `ToPrimitive` без подсказки — то есть метод `@@toPrimitive`, если он вызывается, идет со значением `"default"`. При этом в случае сравнения двух объектов поведение операторов `==` и `===` идентично, и происходит прямое сравнение самих ссылок. В таком случае метод `@@toPrimitive` вообще не вызывается. Подробно приведение и абстрактные операции рассматриваются в книге *Types & Grammar* этой серии.

Символы регулярных выражений

Существуют четыре известных символа, допускающих переопределение для объектов регулярных выражений. Таким образом мы можем выбирать способ их использования четырьмя соответствующими одноименными функциями `String.prototype`.

- `@@match`: Значение регулярного выражения `Symbol.match` представляет собой метод, позволяющий целиком или частично совместить строковое значение с указанным регулярным выражением. Оно используется методом `String.prototype.match(...)`, когда вы передаете в него регулярное выражение в качестве шаблона соответствия.

Алгоритм поиска соответствия, используемый по умолчанию, описан в разделе 21.2.5.6 спецификации ES6 (см. <https://tc39.github.io/ecma262/#sec-regexp.prototype-@@match>). Этот алгоритм можно переопределить, добавив к регулярным выражениям дополнительные функциональные особенности, например возможность ретроспективной проверки.

Символ `Symbol.match` также используется абстрактной операцией `isRegExp` (см. раздел «Функции проверки строки» главы 6), которая определяет, будет ли объект фигурировать в качестве регулярного выражения. Можно принудительно получить отрицательный результат такой проверки, когда нам требуется, чтобы объект не интерпретировался как регулярное выражение. Для этого символу `Symbol.match` присваивается `false` (или значение, которое будет оценено как ложное).

- `@@replace`: Значение регулярного выражения `Symbol.replace` представляет собой метод, используемый в функции `String.prototype.replace(...)` для замены внутри строки одного или всех вхождений последовательности символов, совпадающих с указанным шаблоном регулярного выражения.

Алгоритм замены, используемый по умолчанию, описан в разделе 21.2.5.8 спецификации ES6 (см. <https://tc39.github.io/ecma262/#sec-regexp.prototype-@@replace>). Переопределение этого алгоритма, в частности, предоставляет дополнительные варианты аргумента замены, например поддержку таких выражений, как `"abaca".replace(/a/g, [1, 2, 3])`, дающих в результате `"1b2c3"` благодаря использованию итерируемого объекта для последовательных значений замены.

- `@@search`: Значение регулярного выражения `Symbol.search` представляет собой метод, используемый в функции `String.prototype.search(...)` для поиска внутри строки другой, меньшей строки, совпадающей с заданным регулярным выражением.

Алгоритм поиска, используемый по умолчанию, описан в разделе 21.2.5.9 спецификации ES6 (см. <https://tc39.github.io/ecma262/#sec-regexp.prototype-@@search>).

- `@@split`: Значение регулярного выражения `Symbol.split` представляет собой метод, используемый в функции `String`.

`prototype.split(..)` для разбиения строки на подстроки в месте или местах расположения разделителей, совпадающих с предоставленным регулярным выражением.

Алгоритм разбиения, используемый по умолчанию, описан в разделе 21.2.5.11 спецификации ES6 (см. <https://tc39.github.io/ecma262/#sec-regexp.prototype-@@split>).

Переопределение встроенных алгоритмов регулярных выражений — это занятие не для слабых духом! Движок JS хорошо оптимизирован под работу с регулярными выражениями, соответственно, ваш собственный код, скорее всего, будет выполняться намного медленнее. Данный вид метапрограммирования — аккуратная и мощная техника, но применять ее следует только в случаях, когда это действительно необходимо и оправданно.

Symbol.isConcatSpreadable

Символ `@@isConcatSpreadable` можно определить как булево свойство произвольного объекта (`Symbol.isConcatSpreadable`), например массива или другого итерируемого элемента, указывающее, следует ли *расширить* его, когда он передается методу массива `concat(..)`.

Рассмотрим пример:

```
var    a = [1,2,3],  
      b = [4,5,6];  
  
b[Symbol.isConcatSpreadable] = false;  
  
[].concat( a, b );           // [1,2,3,[4,5,6]]
```

Symbol.unscopables

Символ `@@unscopables` определяется как свойство произвольного объекта (`Symbol.unscopables`), указывающее, какие свойства могут фигурировать в операторе `with` в качестве лексических переменных.

Рассмотрим пример:

```
var    o = { a:1, b:2, c:3 },
      a = 10, b = 20, c = 30;
o[Symbol.unscopables] = {
  a: false,
  b: true,
  c: false
};

with (o) {
  console.log( a, b, c );      // 1 20 3
}
```

Значение `true` в объекте `@@unscopables` указывает, что свойство должно быть скрыто, то есть убрано из переменных лексической области видимости. Значение `false` означает, что его можно включить в перечень этих переменных.



В строгом режиме оператор `with` полностью запрещен. По большому счету, его следует рассматривать как устаревший. Не стоит им пользоваться. Более подробно эта тема раскрывается в книге *Scope & Closures*. А раз оператора `with` рекомендуется избегать, под вопрос ставится и актуальность символа `@@unscopables`.

Прокси

Из появившихся в ES6 функциональных особенностей очевиднее всего относится к метапрограммированию объект `Proxy`.

Это особый вид объекта, который «заключает в себя» другой, обычный объект (или располагается перед ним). Можно зарегистрировать специальные обработчики, также известные как *перехватчики* (traps), которые будут вызываться при выполнении с прокси-объектом различных операций. Эти обработчики в состоянии не только переводить операции на исходный целевой/помещенный в контейнер объект, но и выполнять дополнительные алгоритмы.

Рассмотрим пример *перехватчика*, который можно определить для прокси. Он называется `get` и перехватывает операцию `[[Get]]` при попытке доступа к свойству объекта:

```
var obj = { a: 1 },
    handlers = {
      get(target, key, context) {
        // примечание: target === obj,
        // context === pobj
        console.log( "accessing: ", key );
        return Reflect.get(
          target, key, context
        );
      }
    },
    pobj = new Proxy( obj, handlers );

obj.a;
// 1

pobj.a;
// обращение: a
// 1
```

Мы объявляем обработчик `get(...)` как именованный метод объекта-обработчика (второй аргумент у объекта `Proxy(...)`), получающий ссылку на *целевой объект* (`obj`), имя свойства *key* (`"a"`) и *self*/получатель/прокси (`pobj`).

После оператора трассировки `console.log(...)` мы «направляем» операцию на целевой объект `obj` посредством метода `Reflect.get(...)`. Встроенный объект `Reflect` будет рассматриваться в следующем разделе, пока же запомните, что все доступные перехватчики прокси имеют соответствующую функцию `Reflect` с таким же именем.

Эти отображения сделаны симметричными намеренно. Любой обработчик прокси осуществляет перехват при выполнении соответствующей задачи метапрограммирования, а все вспомогательные объекты `Reflect` выполняют данную задачу с объектом. Каждый обработчик прокси обладает стандартным определением, которое

автоматически обращается к соответствующему объекту `Reflect`. Вы практически гарантированно будете применять объекты `Proxy` и `Reflect` одновременно.

Вот список обработчиков, которые можно задать через прокси для *целевого* объекта/функции, а также информация о том, как и когда они активируются.

`get(..)`

Через `[[Get]]`, доступ к свойству — через прокси-объект (`Reflect.get(..)`, оператор свойства `.` или оператор свойства `[..]`).

`set(..)`

Через `[[Set]]`, значение свойства задается в прокси-объекте (`Reflect.set(..)`, оператор присваивания `=` или деструктурирующее присваивание, если целью является свойство объекта).

`deleteProperty(..)`

Через `[[Delete]]`, свойство удаляется из прокси-объекта (`Reflect.deleteProperty(..)` или `delete`).

`apply(..)` (если цель — функция)

Через `[[Call]]`, прокси-объект вызывается как обычная функция/метод (`Reflect.apply(..)`, `call(..)`, `apply(..)` или оператор вызова `(..)`).

`construct(..)` (если цель — функция конструктора)

Через `[[Construct]]`, прокси-объект вызывается как функция конструктора (`Reflect.construct(..)` или `new`).

`getOwnPropertyDescriptor(..)`

Через `[[GetOwnProperty]]`, из прокси-объекта извлекается дескриптор свойства (`Object.getOwnPropertyDescriptor(..)` или `Reflect.getOwnPropertyDescriptor(..)`).

`defineProperty(..)`

Через `[[DefineOwnProperty]]`, в прокси-объекте задается дескриптор свойства (`Object.defineProperty(..)` или `Reflect.defineProperty(..)`).

`getPrototypeOf(..)`

Через `[[GetPrototypeOf]]`, извлекается `[[Prototype]]` прокси-объекта (`Object.getPrototypeOf(..)`, `Reflect.getPrototypeOf(..)`, `__proto__`, `Object#isPrototypeOf(..)` или `instanceof`).

`setPrototypeOf(..)`

Через `[[SetPrototypeOf]]`, задается `[[Prototype]]` прокси-объекта (`Object.setPrototypeOf(..)`, `Reflect.setPrototypeOf(..)` или `__proto__`).

`preventExtensions(..)`

Через `[[PreventExtensions]]`, прокси-объект делается нерасширяемым (`Object.preventExtensions(..)` или `Reflect.preventExtensions(..)`).

`isExtensible(..)`

Через `[[IsExtensible]]`, проверяется расширяемость прокси-объекта (`Object.isExtensible(..)` или `Reflect.isExtensible(..)`).

`ownKeys(..)`

Через `[[OwnPropertyKeys]]`, извлекается набор собственных свойств и/или собственных символьных свойств прокси-объекта (`Object.keys(..)`, `Object.getOwnPropertyNames(..)`, `Object.getOwnPropertySymbols(..)`, `Reflect.ownKeys(..)` или `JSON.stringify(..)`).

`enumerate(..)`

Через `[[Enumerate]]`, запрашивается итератор для перечислимых собственных и «унаследованных» свойств прокси-объекта (`Reflect.enumerate(..)` или `for...in`).

has(..)

Через `[[HasProperty]]`, проверяется наличие у прокси-объекта собственного или «унаследованного» свойства (`Reflect.has(..)`, `Object#hasOwnProperty(..)` или `"prop"` в `obj`).



Более подробно все эти задачи метапрограммирования будут рассматриваться ниже, в разделе «Reflect API».

В дополнение к перечисленным выше сведениям о действиях, активирующих различные перехватчики, нужно сказать, что некоторые перехватчики активируются косвенным образом через действия по умолчанию какого-то другого перехватчика. Например:

```
var handlers = {
  getOwnPropertyDescriptor(target,prop) {
    console.log(
      "getOwnPropertyDescriptor"
    );
    return Object.getOwnPropertyDescriptor(
      target, prop
    );
  },
  defineProperty(target,prop,desc){
    console.log( "defineProperty" );
    return Object.defineProperty(
      target, prop, desc
    );
  }
},
proxy = new Proxy( {}, handlers );

proxy.a = 2;
// getOwnPropertyDescriptor
// defineProperty
```

Обработчики `getOwnPropertyDescriptor(..)` и `defineProperty(..)` активируются при действии обработчика `set(..)` в момент, когда задается значение свойства (это может быть как добавление ново-

го значения, так и обновление старого). Если вы, кроме того, определите свой собственный обработчик `set(..)`, то по желанию сможете добавить к нему соответствующий контекстный (не целевой!) вызов, запускающий эти перехватчики прокси-объекта.

Пределы возможностей прокси-объектов

Эти обработчики метапрограммирования перехватывают множество фундаментальных операций, применимых к объектам. Однако есть и такие операции, которые (по крайней мере, пока) недоступны для перехвата.

Например, ни одна из следующих операций не перехватывается и не направляется от прокси-объекта `pobj` к целевому объекту `obj`:

```
var    obj = { a:1, b:2 },
      handlers = { .. },
      pobj = new Proxy( obj, handlers );

typeof obj;
String( obj );
obj + "";
obj == pobj;
obj === pobj
```

Возможно, в будущем большее количество фундаментальных операций языка станет доступным для перехвата, что даст нам дополнительные способы расширения JavaScript изнутри.



Существуют различные *инварианты* — поведения, не допускающие переопределения, — применимые к использованию обработчиков прокси. Например, результат обработчика `isExtensible(..)` всегда приводится к типу `boolean`. Инварианты ограничивают возможности настройки поведения через прокси-объекты, но это сделано только для того, чтобы помешать возникновению странных и необычных (или несогласованных) поведения. Условия для инвариантов крайне сложны, поэтому в книге мы их рассматривать не будем, но они достаточно подробно рассмотрены тут: <http://www.2ality.com/2014/12/es6-proxies.html#invariants>.

Отзываемые прокси

Обычный прокси всегда осуществляет прерывание для целевого объекта и не допускает редактирования после своего создания. Пока ссылка на прокси сохраняется, проксирование возможно. Но бывают ситуации, когда требуется прокси-объект, который в какой-то момент следует остановить. Решение нам дает *отзываемый прокси* (revocable proxy).

```
var obj = { a: 1 },
    handlers = {
      get(target, key, context) {
        // примечание: target === obj,
        // context === pobj
        console.log( "accessing: ", key );
        return target[key];
      }
    },
    { proxy: pobj, revoke: prevoke } =
      Proxy.revocable( obj, handlers );

pobj.a;
// обращение: a
// 1

// позднее:
prevoke();

pobj.a;
// TypeError
```

Отзываемый прокси-объект создается методом `Proxy.revocable(..)`, который представляет собой обычную функцию, а не конструктор, как `Proxy(..)`. В остальном же метод принимает все те же два аргумента: *цель* и *обработчики*.

Возвращаемое значение метода `Proxy.revocable(..)` — не сам прокси, как в случае операции `new Proxy(..)`. Вместо этого мы получаем объект с двумя свойствами: `proxy` и `revoke`. Мы воспользовались деструктуризацией объекта (см. раздел «Деструктури-

рующее присваивание» главы 2) и назначили полученные свойства переменным `obj` и `prevoked()` соответственно.

Как только прокси отозван, любые попытки доступа к нему (активация любых его перехватчиков) приведут к ошибке `TypeError`.

Примером использования отзываемого прокси может служить его передача другой части приложения, которая управляет данными в вашей модели, вместо ссылки на реальную модель самого объекта. При изменении или перемещении этого объекта достаточно отключить прокси, и другая часть программы узнает (путем генерации ошибок), что нужно запросить обновленную ссылку на модель.

Применение прокси

Выгоды метапрограммирования объектов `Proxy` очевидны. Мы можем практически полностью перехватывать (и, соответственно, переопределять) поведение объектов, расширяя его за пределы ядра JS. Рассмотрим несколько примеров, демонстрирующих данную возможность.

Прокси в начале, прокси в конце

Как упоминалось выше, обычно считается, что прокси «заклучает в себя» целевой объект. В этом смысле он становится основным объектом, с которым связывается код, а реальный целевой объект остается скрытым/защищенным.

Такие вещи делаются, например, когда нужно передать объект в какое-то место, которому нельзя полностью «доверять», а значит, нужно ввести специальные правила доступа.

Рассмотрим пример:

```
var    messages = [],
      handlers = {
        get(target, key) {
          // строковое значение?
          if (typeof target[key] == "string") {
```

```
        // отфильтровываем пунктуацию
        return target[key]
            .replace( /[^\w]/g, "" );
    }

    // передаем все остальное
    return target[key];
},
set(target, key, val) {
    // задаются только уникальные строки, нижний регистр
    if (typeof val == "string") {
        val = val.toLowerCase();
        if (target.indexOf( val ) == -1) {
            target.push(
                val.toLowerCase()
            );
        }
    }
    return true;
}
},
messages_proxy =
    new Proxy( messages, handlers );

// в другом месте:
messages_proxy.push(
    "heLlo...", 42, "wOrld!!", "WoRld!!"
);

messages_proxy.forEach( function(val){
    console.log(val);
} );
// hello world

messages.forEach( function(val){
    console.log(val);
} );
// hello... world!!
```

Я называю подобное проектирование *прокси в начале* (proxy first), так как сперва мы взаимодействуем именно с прокси.

Мы принудительно ввели специальные правила взаимодействия с переменной `messages_proxy`, которые не распространяются на

саму переменную `messages`. Элементы добавляются только в случае, когда значение уникально и представляет собой строку; кроме того, мы выводим его строчными буквами. Выводя значение объекта `messages_proxu`, мы убираем из строк все знаки препинания.

Этот шаблон можно превратить в диаметрально противоположный, в котором целевой объект будет взаимодействовать с прокси, а не наоборот. В таком случае код работает исключительно с основным объектом. Этот альтернативный вариант проще всего реализуется помещением прокси-объекта в цепочку `[[Prototype]]` основного объекта.

Рассмотрим пример:

```
var    handlers = {
        get(target, key, context) {
            return function() {
                context.speak(key + "!");
            };
        },
        catchall = new Proxy( {}, handlers ),
        greeter = {
            speak(who = "someone") {
                console.log( "hello", who );
            }
        };

// настраиваем 'greeter' на возвращение к 'catchall'
Object.setPrototypeOf( greeter, catchall );
greeter.speak();                                // hello someone
greeter.speak( "world" );                       // hello world

greeter.everyone();                             // hello everyone!
```

Мы взаимодействуем непосредственно с объектом `greeter`, а не с объектом `catchall`. Вызванный метод `speak(..)` принадлежит объекту `greeter` и используется напрямую, но при попытке доступа к такому методу, как `everyone()`, оказывается, что объект `greeter` им не обладает.

По умолчанию свойство объекта должно проверять цепочку `[[Prototype]]` (см. книгу *this & Object Prototypes*), поэтому для свойства `everyone` проверяется объект `catchall`. После этого активизируется обработчик прокси `get()` и возвращает вызывающую метод `speak(..)` функцию с именем свойства, к которому мы обратились (`"everyone"`).

Я называю такой шаблон *прокси в конце* (proxy last), потому что прокси-объект задействуется только при последнем обращении.

«Нет такого свойства/метода»

Разработчики часто жалуются, что по умолчанию объекты не очень хорошо защищены в ситуации, когда имеет место попытка доступа или задания уже не существующего свойства. Вы можете заранее определить все свойства/методы объекта и получить ошибку, когда впоследствии будет использовано несуществующее имя свойства.

Эта задача решается с помощью прокси-объекта. Рассмотрим варианты кода, построенные по принципам *прокси в начале* и *прокси в конце*.

```
var obj = {
  a: 1,
  foo() {
    console.log( "a:", this.a );
  }
},
handlers = {
  get(target, key, context) {
    if (Reflect.has( target, key )) {
      return Reflect.get(
        target, key, context
      );
    }
    else {
      throw "Нет такого свойства/метода!";
    }
  },
}
```

```

        set(target, key, val, context) {
            if (Reflect.has( target, key )) {
                return Reflect.set(
                    target, key, val, context
                );
            }
            else {
                throw "Нет такого свойства/метода!";
            }
        }
    },
    pobj = new Proxy( obj, handlers );

pobj.a = 3;
pobj.foo();                // a: 3

pobj.b = 4;                // Error: Нет такого свойства/метода!
pobj.bar();                // Error: Нет такого свойства/метода!

```

Для методов `get(..)` и `set(..)` операция перенаправляется только в случае существования свойства целевого объекта; в противном случае генерируется ошибка.

Основной объект, с которым должен взаимодействовать код, — прокси (`pobj`), перехватывающий указанные действия, обеспечивая защиту.

Теперь посмотрим, как это реализуется в варианте *прокси в конце*:

```

var handlers = {
    get() {
        throw "Нет такого свойства/метода!";
    },
    set() {
        throw "Нет такого свойства/метода!";
    }
},
pobj = new Proxy( {}, handlers ),
obj = {
    a: 1,
    foo() {
        console.log( "a:", this.a );
    }
};

```

```
    }  
};  
  
// настраиваем 'obj' вернуться к 'pobj'  
Object.setPrototypeOf( obj, pobj );  
  
obj.a = 3;  
obj.foo();           // a: 3  
  
obj.b = 4;           // Error: Нет такого свойства/метода!  
obj.bar();           // Error: Нет такого свойства/метода!
```

В том, что касается определения обработчика, вариант кода *прокси в конце* получится значительно более простым. Вместо перехвата операций `[[Get]]` и `[[Set]]` и перенаправления их только в том случае, когда целевое свойство существует, мы положились на то, что если операция `[[Get]]` или `[[Set]]` добралась до нашего запасного варианта `pobj`, значит, она уже прошла всю цепочку `[[Prototype]]` и не обнаружила соответствующего свойства. Следовательно, мы можем сгенерировать ошибку, не прибегая к условным конструкциям. Здорово, да?

Прокси, взламывающий цепочку `[[Prototype]]`

Основной канал активизации механизма `[[Prototype]]` — операция `[[Get]]`. Если непосредственно у объекта свойство не обнаруживается, операция `[[Get]]` автоматически переходит к объекту `[[Prototype]]`.

Это означает, что перехватчик метода `get(...)` у прокси можно применять для эмуляции или расширения концепции механизма `[[Prototype]]`.

В качестве первого взлома мы рассмотрим создание двух взаимосвязанных через `[[Prototype]]` объектов (или, по крайней мере, выглядящих таковыми). На самом деле создать зацикленную цепочку `[[Prototype]]` нельзя, так как движок сгенерирует сообщение об ошибке. Но прокси-объект сможет это эмулировать!

Рассмотрим пример:

```
var handlers = {
  get(target, key, context) {
    if (Reflect.has( target, key )) {
      return Reflect.get(
        target, key, context
      );
    }
    // имитация зацикленного '[[Prototype]]'
    else {
      return Reflect.get(
        target[
          Symbol.for( "[[Prototype]]" )
        ],
        key,
        context
      );
    }
  },
};
obj1 = new Proxy(
  {
    name: "obj-1",
    foo() {
      console.log( "foo:", this.name );
    }
  },
  handlers
);
obj2 = Object.assign(
  Object.create( obj1 ),
  {
    name: "obj-2",
    bar() {
      console.log( "bar:", this.name );
      this.foo();
    }
  }
);

// имитация зацикленной ссылки '[[Prototype]]'
obj1[ Symbol.for( "[[Prototype]]" ) ] = obj2;
```

```
obj1.bar();  
// bar: obj-1 <-- через прокси, имитирующий [[Prototype]]  
// foo: obj-1 <-- контекст 'this' все еще сохранен  
  
obj2.foo();  
// foo: obj-2 <-- через [[Prototype]]
```



В этом примере нам не нужно использовать прокси / перенаправлять операцию `[[Set]]`, поэтому мы реализовали все достаточно просто. Для полностью совместимой эмуляции `[[Prototype]]` нужно реализовать обработчик `set(..)`, ищущий в цепочке `[[Prototype]]` совпадающее свойство и уважающий поведение его дескриптора (например, `set`, `writable`). См. книгу *this & Object Prototypes*.

В предыдущем фрагменте объект `obj2` — это `[[Prototype]]`, связанный с `obj1` посредством оператора `Object.create(..)`. Для формирования обратной (циклической) ссылки мы создаем свойство у объекта `obj1` в месте расположения символа `Symbol.for("[[Prototype]]")` (см. раздел «Символы» главы 2). Он может выглядеть как особый/магический, но в действительности таковым не является. Он всего лишь позволяет мне удобно именовать хук, который выглядит семантически связанным с выполняемой мной задачей.

Затем обработчик `get(..)` прокси-объекта первым делом ищет в нем запрошенный ключ. Если таковой не обнаруживается, операция вручную передается ссылке на объект, хранящейся в местоположении цели `Symbol.for("[[Prototype]]")`.

Важное преимущество этого шаблона — тот факт, что определения объектов `obj1` и `obj2` по большей части не затронуты настройкой между ними циклической связи. Хотя в предыдущем фрагменте кода все этапы были связаны между собой ради краткости, при ближайшем рассмотрении оказывается, что логическая схема обработчика прокси целиком обобщенная — то есть она не знает ничего конкретного об объекте `obj1` или `obj2`. Поэтому ее можно поместить в связывающий их простой вспомогательный объект,

например `setCircularPrototypeOf(..)`. Я оставляю это как упражнение читателям.

Теперь, когда вы знаете, как с помощью обработчика `get(..)` эмулировать связь `[[Prototype]]`, продвинемся в своем взломе чуть дальше. Как насчет того, чтобы заменить циклическую ссылку `[[Prototype]]` множественными связями `[[Prototype]]` (такое явление известно как множественное наследование)? Сделать это достаточно просто:

```
var obj1 = {
  name: "obj-1",
  foo() {
    console.log( "obj1.foo:", this.name );
  },
},
obj2 = {
  name: "obj-2",
  foo() {
    console.log( "obj2.foo:", this.name );
  },
  bar() {
    console.log( "obj2.bar:", this.name );
  }
},
handlers = {
  get(target, key, context) {
    if (Reflect.has( target, key )) {
      return Reflect.get(
        target, key, context
      );
    }
    // имитируем множественный '[[Prototype]]'
    else {
      for (var P of target[
        Symbol.for( "[[Prototype]]" )
      ]) {
        if (Reflect.has( P, key )) {
          return Reflect.get(
            P, key, context
          );
        }
      }
    }
  }
}
```

```
        }
    }
},
obj3 = new Proxy(
    {
        name: "obj-3",
        baz() {
            this.foo();
            this.bar();
        }
    },
    handlers
);

// имитируем множественные ссылки '[[Prototype]]'
obj3[ Symbol.for( "[[Prototype]]" ) ] = [
    obj1, obj2
];

obj3.baz();
// obj1.foo: obj-3
// obj2.bar: obj-3
```



Как упоминалось в примечании после предыдущего примера циклического `[[Prototype]]`, мы не реализовывали обработчик `set(...)`, но это потребуется для получения полного решения, эмулирующего операции `[[Set]]` в том виде, в котором они присутствуют в поведении обычных объектов `[[Prototype]]`.

Настройки объекта `obj3` превратили его в делегат, посылающий сообщения объектам `obj1` и `obj2`. В методе `obj3.baz()` вызов `this.foo()` заканчивается извлечением `foo()` из объекта `obj1` (первым пришел, первым обслужен, несмотря на наличие `foo()` у `obj2`). Если поменять порядок связывания на `obj2`, `obj1`, будет обнаружен и использован вариант `obj2.foo()`.

Но так как вызов `this.bar()` не видит метода `bar()` у объекта `obj1`, он начинает проверять объект `obj2`, в котором и находит соответствие.

Объекты `obj1` и `obj2` представляют две параллельные цепочки `[[Prototype]]` объекта `obj3`. Объекты `obj1` и/или `obj2` сами по себе могут обычным образом делегировать `[[Prototype]]` другим объектам или даже выступать в роли прокси-объекта, посылающего сообщения нескольким делегатам (именно таким объектом является `obj3`).

Как в более раннем примере с циклической связью `[[Prototype]]`, определения `obj1`, `obj2` и `obj3` практически полностью отделены от общей логической схемы прокси, обрабатывающей отправку сообщений набору делегатов. Не составляет труда определить служебную функцию `setPrototypesOf(..)` (обратите внимание на букву «s» в конце, это множественное число!), принимающую в качестве аргумента основной объект и список объектов, с которыми нужно симитировать связи типа `[[Prototype]]`. Это упражнение я также оставляю для вашей самостоятельной работы.

Надеюсь, что примеры убедительно показали вам, какой мощный инструмент представляют собой прокси-объекты. Существует и множество задач метапрограммирования, выполнение которых стало возможным именно благодаря прокси.

Reflect API

`Reflect` представляет собой обычный объект (такой же, как, например, объект `Math`), а не функцию/конструктор, как другие встроенные элементы.

С ним связаны статические функции, позволяющие выполнять различные задачи метапрограммирования. Они имеют однозначные соответствия среди методов-обработчиков (*перехватчиков*), определенных для прокси-объектов.

Некоторые из них могут показаться вам знакомыми, так как совпадают с одноименными методами класса `Object`:

- `Reflect.getOwnPropertyDescriptor(..);`
- `Reflect.defineProperty(..);`

- `Reflect.getPrototypeOf(..);`
- `Reflect.setPrototypeOf(..);`
- `Reflect.preventExtensions(..);`
- `Reflect.isExtensible(..).`

Эти методы в общем случае ведут себя так же, как их аналоги `Object.*`. Но есть одно отличие. Методы `Object.*` пытаются привести свой первый аргумент (целевой объект) к объекту, если тот пока им не является. Методы `Reflect.*` в подобных случаях просто генерируют сообщение об ошибке.

Доступ к ключам объекта и их анализ осуществляются следующими методами.

`Reflect.ownKeys(..)`

Возвращает список всех собственных ключей (не «унаследованных»), как методы `Object.getOwnPropertyNames(..)` и `Object.getOwnPropertySymbols(..)`. См. следующий раздел «Порядок свойств».

`Reflect.enumerate(..)`

Возвращает итератор, который производит набор всех ключей, не являющихся символьными (собственных и «унаследованных»), но при этом *перечислимых* (см. книгу *this & Object Prototypes*). По сути, этот набор ключей совпадает с тем, который обрабатывается циклом `for...in`. Информацию о порядке следования ключей см. в разделе «Порядок свойств».

`Reflect.has(..)`

По сути, представляет собой оператор, проверяющий, принадлежит свойство объекту или цепочке `[[Prototype]]` этого объекта. Например, метод `Reflect.has(o, "foo")` проверяет наличие строки `"foo"` в объекте `o`.

Вызовы функций и запуск конструктора можно выполнять и вручную, независимо от обычного синтаксиса (например, `(..)` и `new`) их использования.

`Reflect.apply(...)`

Так, запись `Reflect.apply(foo, thisObj, [42, "bar"])` вызывает функцию `foo(...)` с `thisObj` на месте ключевого слова `this`, передавая в нее в качестве аргументов значения 42 и "bar".

`Reflect.construct(...)`

Например, запись `Reflect.construct(foo, [42, "bar"])`, по сути, вызывает оператор `new foo(42, "bar")`.

Вручную можно также выполнять доступ к свойствам объекта, их настройку и удаление. Для этого применяются следующие методы.

`Reflect.get(...)`

Так, запись `Reflect.get(o, "foo")` извлекает свойство `o.foo`.

`Reflect.set(...)`

Например, запись `Reflect.set(o, "foo", 42)` выполняет присваивание `o.foo = 42`.

`Reflect.deleteProperty(...)`

Например, запись `Reflect.deleteProperty(o, "foo")` удаляет свойство `o.foo`.

Возможности метапрограммирования, возникающие благодаря объекту `Reflect`, дают нам программные эквиваленты для эмуляции различных синтаксических функциональных особенностей, которые раньше были представлены только в виде скрытых абстрактных операций. Это позволяет нам, в частности, расширять функциональность и различные API для *предметно-ориентированных языков* (DSL — domain specific languages).

Порядок свойств

До ES6 порядок перечисления ключей/свойств объекта зависел от реализации и в спецификации не оговаривался. В общем случае большинство движков перечисляло их в той последовательности,

в которой они были созданы, хотя разработчикам крайне не рекомендовалось полагаться на этот порядок.

В ES6 определили порядок перечисления собственных свойств (спецификация ES6, раздел 9.1.12) через алгоритм `[[OwnPropertyKeys]]`, который выводит эти свойства (строки или символы), независимо от их перечислимости. Порядок гарантирован только для метода `Reflect.ownKeys(..)` (и, следовательно, для методов `Object.getOwnPropertyNames(..)` и `Object.getOwnPropertySymbols(..)`).

Он определяется следующим образом.

1. Первым делом в возрастающем порядке нумеруются любые собственные свойства, являющиеся целыми индексами.
2. Затем в порядке создания нумеруются имена остальных собственных строковых свойств.
3. Наконец, в порядке создания нумеруются собственные символные свойства.

Рассмотрим пример:

```
var o = {};  
  
o[Symbol("c")] = "yay";  
o[2] = true;  
o[1] = true;  
o.b = "awesome";  
o.a = "cool";  
  
Reflect.ownKeys( o );           // [1, 2, "b", "a", Symbol(c)]  
Object.getOwnPropertyNames( o ); // [1, 2, "b", "a"]  
Object.getOwnPropertySymbols( o ); // [Symbol(c)]
```

В то же самое время алгоритм `[[Enumerate]]` (спецификация ES6, раздел 9.1.11) порождает только перечислимые свойства из целевого объекта и его цепочки `[[Prototype]]`. Они используются как методом `Reflect.enumerate(..)`, так и циклом `for...in`. Наблюдаемый порядок зависит от конкретной реализации и спецификацией не контролируется.

В противоположность этому метод `Object.keys(...)` задействует алгоритм `[[OwnPropertyKeys]]` для получения полного списка собственных ключей. Он отфильтровывает все свойства, не являющиеся перечислимыми, после чего меняет их порядок для совместимости с унаследованным зависимым от реализации поведением, особенно в случае метода `JSON.stringify(...)` и цикла `for...in`. Таким образом, в обобщенном смысле порядок также совпадает с задаваемым методом `Reflect.enumerate(...)`.

Другими словами, все четыре механизма (метод `Reflect.enumerate(...)`, метод `Object.keys(...)`, цикл `for...in` и метод `JSON.stringify(...)`) дадут нам один и тот же зависящий от конкретной реализации порядок, хотя с технической точки зрения результат будет достигаться различными способами.

Реализации с помощью четырех описанных механизмов могут обеспечивать порядок, совпадающий с тем, который задает алгоритм `[[OwnPropertyKeys]]`, но это не обязательное условие. Так или иначе, вы, скорее всего, будете наблюдать следующий вариант упорядоченности:

```
var o = { a: 1, b: 2 };
var p = Object.create( o );
p.c = 3;
p.d = 4;

for (var prop of Reflect.enumerate( p )) {
  console.log( prop );
}
// c d a b

for (var prop in p) {
  console.log( prop );
}
// c d a b

JSON.stringify( p );
// {"c":3,"d":4}

Object.keys( p );
// ["c","d"]
```

Подведем итоги: спецификация ES6 гарантирует предсказуемый и надежный порядок только в случае методов `Reflect.ownKeys(...)`, `Object.getOwnPropertyNames(...)` и `Object.getOwnPropertySymbols(...)`. В этих случаях безопасно создавать код, полагающийся на установленный порядок.

Методы `Reflect.enumerate(...)`, `Object.keys(...)`, а также цикл `for...in` (равно как и метод `JSON.stringify(...)` в обобщенном смысле) все еще имеют общий наблюдаемый порядок, как и раньше. Но он не должен совпадать с отображаемым методом `Reflect.ownKeys(...)`, и использовать зависящий от реализации порядок до сих пор следует с осторожностью.

Тестирование функциональных особенностей

Что такое тестирование функциональных особенностей? Это проверка, которую вы делаете, чтобы определить, доступна какая-либо из них или нет. Иногда проверяется не только ее наличие, но и согласованность с определенным поведением — функциональные особенности могут существовать, но работать некорректно.

Это техника метапрограммирования, позволяющая проверить, в какой среде запускается ваша программа, чтобы затем определить, как должен вести себя код.

Чаще всего такое тестирование применяется в JS для проверки существования какого-либо API и, если такового нет, для определения полизаполнения (см. главу 1). Например:

```
if (!Number.isNaN) {  
  Number.isNaN = function(x) {  
    return x !== x;  
  };  
}
```

Оператор `if` в данном случае связан с метапрограммированием: мы проверяем нашу программу и среду ее выполнения, чтобы определить, стоит ли продолжать работу.

А как происходит тестирование функциональных особенностей, связанных с новым синтаксисом? Можно было бы попробовать, например, такой вариант:

```
try    {  
    a = () => {};  
    ARROW_FUNCS_ENABLED = true;  
}  
catch (err) {  
    ARROW_FUNCS_ENABLED = false;  
}
```

К сожалению, это не работает, так как JS-программы — компилируемые. Соответственно, если движок еще не поддерживает появившиеся в ES6 стрелочные функции, он не сможет обработать запись `() => {}`. Наличие синтаксической ошибки останавливает работу программы, что мешает ей впоследствии по-разному отвечать на вопрос, поддерживается ли тестируемая функция.

Для написания метапрограммы, тестирующей функциональные особенности, связанные с синтаксисом, нам нужен способ изолировать процедуру тестирования от начального этапа компиляции, через который проходит программа. Например, если предназначенный для тестирования код сохранить в виде строки, движок JS по умолчанию не должен будет компилировать ее, пока не получит явную команду.

Наверное, в этот момент вы подумали о том, что имеет смысл воспользоваться методом `eval(..)?`

Не торопитесь. Если вы прочтете книгу *Scope & Closures* серии *You Don't Know JS*, вы узнаете, почему применение этого метода — не очень хорошая идея. Но существует альтернативный вариант с меньшим количеством недостатков: конструктор `Function(..)`.

Рассмотрим пример:

```
try    {  
    new Function( "( () => {} )" );  
    ARROW_FUNCS_ENABLED = true;  
}  
catch (err) {  
    ARROW_FUNCS_ENABLED = false;  
}
```

Итак, теперь наше метапрограммирование сводится к тому, что мы определяем, компилируется ли конкретным движком некая функциональная особенность, например стрелочные функции. Вполне вероятно, вы уже задаете себе вопрос, как распорядиться этой информацией.

При наличии проверок для различных API и определений полизаполнений можно предусмотреть конкретный вариант действий, как на случай успеха тестирования, так и на случай неудачи. Но что мы можем сделать с информацией о том, что переменная `ARROW_FUNCS_ENABLED` имеет значение `true` или `false`?

Если движок не поддерживает некую функциональную особенность, соответствующий синтаксис попросту не появится в файле. Мы не можем взять и вставить в файл две функции, одна из которых включает рассматриваемый синтаксис, а вторая нет.

Однако тестирование позволяет определить, какой из набора JS-файлов следует загрузить. Например, если набор тестов для функциональных особенностей помещен в загрузчик вашего JS-приложения, можно проверить среду и узнать, допустима прямая загрузка и запуск или же требуется транскомпилированная версия кода (см. главу 1).

Эта техника называется *разделенной поставкой* (split delivery).

Здесь учитывается тот факт, что соответствующие стандарту ES6 JS-программы иногда могут запускать в полностью «исходном формате» браузеры, поддерживающие ES6+, но бывают случаи,

когда требуется транскомпиляция кода для запуска в браузерах, работающих с более ранними версиями стандарта. Если все время загружать и использовать транскомпилированный код, даже в новых, совместимых с ES6 средах, по крайней мере, часть времени работа этого кода будет далека от оптимальной. А такого лучше избегать. При большей сложности и продуманности разделенная поставка — более совершенный и надежный подход, заполняющий зазор между тем кодом, который вы пишете, и наличием поддержки функциональных особенностей в браузерах, где запускаются программы.

FeatureTests.io

Создание тестов для функциональных особенностей всего синтаксиса ES6+, а также для семантических поведений представляет собой крайне сложную задачу, которую вы вряд ли захотите решать самостоятельно. Поскольку такие тесты требуют динамической компиляции (`new Function(...)`), имеет место досадная потеря производительности.

Кроме того, проведение этих тестов при каждом запуске приложения — по большому счету пустая трата ресурсов. Ведь в среднем браузер пользователя обновляется раз в несколько недель, и далеко не каждое обновление сопровождается вводом новых функциональных особенностей.

Наконец, управление списком функциональных тестов, применяемых к конкретной основе кода — ведь ES6 во всей его полноте программы практически никогда не используют, — это неуправляемый и ненадежный процесс.

И здесь вам поможет служба FeatureTests.io (<https://featuretests.io/>).

Ее библиотеку можно загрузить к себе на страницу и затем подгружать новейшие определения тестов, чтобы были доступны все варианты функционального тестирования. По возможности оно выполняется в режиме фоновой обработки с использованием

средства Web Workers для уменьшения издержек производительности. Также применяется средство долговременного хранения LocalStorage, кэширующее результаты таким образом, чтобы их можно было использовать на всех посещаемых вами сайтах, где задействована данная служба. Это резко уменьшает объем тестирования, необходимого для каждого экземпляра браузера.

Таким образом вы сможете проводить функциональное тестирование на лету в браузерах всех пользователей и, динамически применяя результаты тестов, предоставлять код, наиболее подходящий под среду конкретного пользователя (ни больше ни меньше).

Кроме того, служба предоставляет вам инструменты и API для сканирования файлов, чтобы определить, какие именно функции вам нужны, а это дает возможность полностью автоматизировать процесс сборки в режиме разделенной поставки.

Служба FeatureTests.io позволяет использовать функциональное тестирование для всех частей стандарта ES6 и следующих, гарантируя, что для любой среды будет загружаться только наиболее подходящий код.

Оптимизация хвостовой рекурсии

Обычно при вызове одной функции из другой для отдельного управления переменными и состоянием этого второго вызова выделяется второй *стековый кадр* (stack frame), что увеличивает время работы и требует дополнительной памяти.

Цепочка стека вызовов обычно совершает максимум 10–15 переходов от одной функции к другой. В таких сценариях работа с памятью не составляет какой-либо проблемы.

Но если рассмотреть рекурсивное программирование (когда функция раз за разом вызывает сама себя) или взаимную рекурсию с двумя или более функциями, вызывающими друг друга, глубина стека вызова может легко составить сотни, а то и тысячи уровней.

Скорее всего, вы понимаете, какую проблему создает неограниченный рост потребления памяти.

Движкам JavaScript приходится устанавливать некий предел, чтобы написанные с применением подобной техники программы не «падали» из-за того, что браузеру и устройству перестало хватать памяти. При достижении лимита мы видим досадную ошибку «RangeError: Maximum call stack size exceeded» (превышен максимальный размер стека вызова).



Предельная глубина стека вызова в спецификации не оговаривается. Она зависит от конкретной реализации и различается в разных браузерах и на разных устройствах. Никогда не пишите код, исходя из предположения о наблюдаемых лимитах, так как они могут меняться.

Существуют шаблоны вызова функций, называемые *хвостовыми вызовами* (tail calls), которые можно оптимизировать, избежав выделения дополнительных стековых кадров. А раз так, нет никаких оснований как-либо ограничивать глубину стека вызовов, и, соответственно, движки могут запускать такие функции без ограничений.

Хвостовой вызов представляет собой оператор `return` в вызове функции, после которого не должно происходить ничего, кроме возврата значения. Такая оптимизация применима только в строгом режиме. Вот вам еще одна причина придерживаться именно этого режима написания кода!

Давайте рассмотрим пример вызова функции, который находится *не* в хвостовой позиции:

```
"use strict";
```

```
function foo(x) {  
    return x * 2;  
}
```

```
function bar(x) {  
    // не хвостовая рекурсия  
    return 1 + foo( x );  
}  
  
bar( 10 );                // 21
```

После завершения вызова `foo(x)` необходимо выполнить еще и операцию `1 + ...`, соответственно, нам приходится сохранять состояние вызова `bar(...)`.

Следующий пример демонстрирует вызовы функций `foo(...)` и `bar(...)`, когда *оба* находятся в хвостовой позиции, другими словами, они — последнее, что происходит в их ветви кода (не считая оператора `return`):

```
"use strict";  
  
function foo(x) {  
    return x * 2;  
}  
  
function bar(x) {  
    x = x + 1;  
    if (x > 10) {  
        return foo( x );  
    }  
    else {  
        return bar( x + 1 );  
    }  
}  
  
bar( 5 );                // 24  
bar( 15 );               // 32
```

В этой программе вызов функции `bar(...)` рекурсивный, в то время как вызов `foo(...)` обычный. В обоих случаях вызовы функций находятся в *корректной хвостовой позиции*. Выражение `x + 1` вычисляется до вызова `bar(...)`, и когда бы этот вызов ни завершился, произойдет только выполнение оператора `return`.

Корректные хвостовые вызовы в подобной форме допускают оптимизацию, после которой выделение дополнительного стекового кадра уже не требуется. Вместо этого движок повторно использует уже существующий кадр. Данный механизм работает, функции не нужно сохранять текущее состояние, ведь после корректного хвостового вызова ничего не происходит.

На практике оптимизация хвостовой рекурсии означает отсутствие ограничений на глубину стека вызовов. Этот прием слегка улучшает обычные вызовы функций в обычных программах и, что более важно, дает возможность использовать рекурсию для программных выражений даже в ситуациях, когда глубина стека вызова составляет десятки тысяч уровней.

Мы больше не будем теоретизировать на тему применения рекурсии для решения задач. Вместо этого вы начнете использовать ее в реальных программах!

С точки зрения ES6 все корректные хвостовые вызовы должны оптимизироваться именно таким способом, вне зависимости от наличия в них рекурсии.

Перезапись хвостового вызова

Проблема состоит в том, что оптимизация доступна только для корректных хвостовых вызовов; все прочие, разумеется, будут работать, но не перестанут требовать выделения стекового кадра. Чтобы оптимизация прошла успешно, следует проявить осторожность при структурировании функций с корректными хвостовыми вызовами.

Если ваша функция не попадает под это определение, для оптимизации может потребоваться вручную переписать код.

Рассмотрим пример:

```
"use strict";  
  
function foo(x) {  
    if (x <= 1) return 1;
```

```
    return (x / 2) + foo( x - 1 );  
}  
  
foo( 123456 );           // RangeError
```

Вызов `foo(x-1)` не относится к корректным хвостовым вызовам, так как результат работы этой функции перед возвращением следует добавить к $(x / 2)$.

Можно переписать код следующим образом, чтобы сделать доступным для оптимизации в движке, поддерживающем ES6:

```
"use strict";  
  
var foo = (function(){  
    function _foo(acc,x) {  
        if (x <= 1) return acc;  
        return _foo( (x / 2) + acc, x - 1 );  
    }  
  
    return function(x) {  
        return _foo( 1, x );  
    };  
})();  
  
foo( 123456 );           // 3810376848.5
```

При запуске предыдущего фрагмента в среде ES6, реализующей оптимизацию хвостовой рекурсии, вы получите показанный выше ответ 3810376848.5. Но в случае движков, не допускающих этого вида оптимизации, вы все равно будете сталкиваться с ошибкой `RangeError`.

Другие варианты оптимизации

Есть и другие техники переписывания кода, позволяющие избежать увеличения стека при каждом следующем вызове.

Одна из них называется *трамплином* (trampolining) и сводится к тому, что каждый частичный результат представляется в виде

функции, которая возвращает или другую частично результирующую функцию, или окончательный результат. Нужно только поместить все это в цикл, который остановится только после того, как вы перестанете получать функцию.

Рассмотрим пример:

```
"use strict";

function trampoline( res ) {
  while (typeof res == "function") {
    res = res();
  }
  return res;
}

var foo = (function(){
  function _foo(acc,x) {
    if (x <= 1) return acc;
    return function partial(){
      return _foo( (x / 2) + acc, x - 1 );
    };
  }

  return function(x) {
    return trampoline( _foo( 1, x ) );
  };
})();
foo( 123456 ); // 3810376848.5
```

Нам требуются минимальные изменения, чтобы превратить рекурсию в цикл с использованием метода `trampoline(...)`.

1. Первым делом мы поместим строчку `return _foo ..` в функциональное выражение `return partial() { ...`.
2. После этого вставим вызов функции `_foo(1,x)` в вызов метода `trampoline(...)`.

Причина, по которой данная техника не страдает от ограничений на размер стека вызовов, связана с тем, что каждая из внутренних функций `partial(...)` возвращает результат внутри цикла `while` в методе `trampoline(...)`, который ее запускает. После этого проис-

ходит следующая итерация цикла. Другими словами, функция `partial(...)` не вызывает рекурсивно сама себя, а всего лишь возвращает другую функцию. Глубина стека остается неизменной, поэтому функция может работать так долго, как требуется.

Реализованная таким способом техника трамплина использует замыкание внутренней функции `partial()` на переменных `x` и `acc`, которое обеспечивает сохранение состояния между итерациями. Преимущество такого подхода состоит в том, что логическая схема цикла помещена в служебную функцию многократного использования `trampoline(...)`, версии которой содержатся во многих библиотеках. Вы можете вставлять ее в свои программы произвольное число раз с различными алгоритмами трамплинов.

Разумеется, для действительно глубокой оптимизации (когда многократное использование не является целью) можно отказаться от замыкания и поместить отслеживание состояния переменной `acc` в одну область видимости с циклом. Эта техника называется *разворачиванием рекурсии* (recursion unrolling).

```
"use strict";

function foo(x) {
    var    acc = 1;
    while (x > 1) {
        acc = (x / 2) + acc;
        x = x - 1;
    }
    return acc;
}

foo( 123456 );                                // 3810376848.5
```

Такая реализация алгоритма проще читается и, скорее всего, будет работать лучше любой из изученных нами форм. Может показаться, что это однозначный лидер, и непонятно, зачем вообще применяются остальные подходы.

Но есть причины, по которым далеко не всегда следует вручную разворачивать рекурсию.

- Вместо факторизации логической схемы трамплина (цикла) для ее повторного использования мы ее встроили. Это замечательно работает при наличии всего одного такого элемента, но когда он встречается в программе десяток раз, скорее всего, вам потребуется возможность повторного использования, чтобы сделать код более коротким и управляемым.
- В данном случае намеренно приведен достаточно простой пример, чтобы проиллюстрировать различные формы. На практике же существует множество более сложных алгоритмов, например взаимная рекурсия (когда несколько функций вызывают друг друга).

Чем глубже мы погружаемся в тему, тем более неавтоматическими и запутанными становятся варианты оптимизации с *развертыванием*. Кажущиеся преимущества, такие как легкая читаемость, быстро отходят на второй план. Основная польза от рекурсии состоит в том, что даже когда функция находится в корректной хвостовой позиции, рекурсия сохраняет алгоритм читаемым и снижает расходы движка на оптимизацию производительности.

При написании алгоритмов с функциями в корректной хвостовой позиции движок ES6 применяет оптимизацию хвостовой рекурсии, обеспечивая работу кода с постоянной глубиной стека (повторным использованием стековых кадров). За счет рекурсии в этом случае вы получаете читабельность и большинство преимуществ в сфере производительности, причем без ограничений на продолжительность работы.

Метaproгpаммирование?

Как оптимизация хвостовой рекурсии связана с метaproгpаммированием?

В разделе «Тестирование функциональных особенностей» вы узнали, что во время выполнения можно указывать, какие именно

функциональные особенности поддерживает движок. Это касается и оптимизации хвостовых вызовов, хотя выполняется она достаточно грубыми методами. Рассмотрим пример:

```
"use strict";

try {
  (function foo(x){
    if (x < 5E5) return foo( x + 1 );
  })( 1 );

  TCO_ENABLED = true;
}
catch (err) {
  TCO_ENABLED = false;
}
```

В случае движка, не поддерживающего оптимизацию хвостовой рекурсии, цикл в итоге прекратит свою работу с исключением, перехваченным блоком `try...catch`. В противном случае цикл легко завершится благодаря оптимизации. Не очень красиво, да?

Но как метапрограммирование, связанное с оптимизацией хвостовой рекурсии (точнее, с ее отсутствием), помогает нашему коду? Самый простой ответ на этот вопрос — функциональное тестирование, позволяющее решить, какая из версий кода приложения будет загружена: использующая рекурсию или альтернативная, преобразованная/транскомпилированная, для которой рекурсия не требуется.

Самонастраивающийся код

Впрочем, на проблему можно взглянуть и под другим углом:

```
"use strict";

function foo(x) {
  function _foo() {
    if (x > 1) {
      acc = acc + (x / 2);
    }
  }
}
```

```
        x = x - 1;
        return _foo();
    }

    }

    var acc = 1;

    while (x > 1) {
        try {
            _foo();
        }
        catch (err) { }
    }

    return acc;
}

foo( 123456 );                                // 3810376848.5
```

Этот алгоритм пытается сделать максимальную часть работы с рекурсией, параллельно отслеживая ход выполнения через переменные `x` и `acc`. Если задача целиком решается с помощью рекурсии — хорошо. Если же в какой-то момент движок прерывает ее, мы перехватываем событие с помощью блока `try..catch` и пытаемся начать снова с прерванного места.

Я рассматриваю это как форму метапрограммирования, потому что во время выполнения вы проверяете способность движка полностью (рекурсивно) завершить задачу и обойти любые (не связанные с оптимизацией хвостовой рекурсии) ограничения, которые могут помешать.

Готов биться об заклад, что на первый (или даже на второй!) взгляд этот код покажется вам более неприглядным, чем предыдущие версии. Кроме того, он гораздо медленнее работает (на более длинных прогонах в средах без оптимизации хвостовой рекурсии).

Основным его преимуществом, кроме возможности завершать задачи произвольного размера в случае движков без оптимизации хвостовой рекурсии, является тот факт, что этот вариант обхода

связанных с рекурсией ограничений стека своей гибкостью намного превосходит технику трамплина или развертывание вручную.

По сути, в данном случае `_foo()` — своего рода заместитель практически любой рекурсивной задачи, даже взаимной рекурсии. Все остальное представляет собой шаблон, который должен работать практически со всеми алгоритмами.

Единственной «проблемой» является тот факт, что для возможности возобновления после достижения лимита рекурсии состояние рекурсии должно храниться в переменных с ограниченной областью видимости, существующих вне рекурсивной функции или функций. Такой эффект мы получаем, оставив переменные `x` и `acc` вне функции `_foo()`, вместо того чтобы передать их в эту функцию в качестве аргументов, как делали ранее.

К такому способу работы можно адаптировать практически любой рекурсивный алгоритм. Это наиболее широко применимый способ использования оптимизации хвостовой рекурсии с минимальным переписыванием.

Здесь все еще используется корректное хвостовое положение, то есть код *прогрессивно улучшается*, начиная с многократного использования цикла в более старых браузерах и до полной оптимизации хвостовой рекурсии в средах ES6+. Мне кажется, это очень здорово!

Подводим итоги

Метапрограммирование — это ситуация, в которой вы заставляете логическую схему программы сконцентрироваться на ней самой (или на среде исполнения), чтобы проанализировать собственную структуру или отредактировать ее. Основная ценность метапрограммирования — расширение обычных механизмов языка, предоставляющее дополнительные возможности.

И до появления ES6 язык JavaScript вполне позволял заниматься метапрограммированием, но последний стандарт существенно расширил этот процесс, добавив несколько функциональных особенностей.

Новые возможности — от определения имен анонимных функций до метасвойств, рассказывающих о способах вызова конструктора, — помогают анализировать структуру работающей программы лучше, чем когда-либо раньше. Известные символы позволяют переопределять встроенные поведения, например приведение объекта к примитивному значению. Прокси дают возможность прерывать и настраивать различные низкоуровневые операции с объектами, а объект **Reflect** предоставляет средства их эмуляции.

Функциональное тестирование даже для таких незаметных семантических поведений, как оптимизация хвостовой рекурсии, сдвигает фокус метапрограммирования с кода на возможности самого движка JS. Если программы лучше понимают, что способна дать среда исполнения, то они самостоятельно настраиваются на оптимальный способ работы.

Следует ли вам заниматься метапрограммированием? Я советую первым делом сосредоточиться на изучении способов работы основных механизмов языка. Как только вы полностью освоитесь с тем, что способен дать JS сам по себе, придет время использовать мощные возможности метапрограммирования для расширения доступных вам горизонтов!

8

За пределами ES6

На момент написания этой книги окончательный проект ES6 (*ECMAScript 2015*) скоро будет направлен на официальное голосование для одобрения ECMA. Но уже сейчас, несмотря на то что ES6 находится только на стадии завершения, комитет TC39 приступил к работе над функциональными особенностями для стандарта ES7/2016 и следующих за ним.

Как обсуждалось в главе 1, ожидается, что темп прогресса для JS ускорится, перейдя от обновлений раз в несколько лет к появлению официальных новых версий раз в год (вот почему появилась идея именования на основе года). Это радикально изменит подход к изучению языка и поддержанию уровня знаний.

Но куда важнее тот факт, что комитет собирается работать над отдельными функциональными особенностями. Как только спецификация какой-либо из них оказывается готовой, а проблемные места — проработанными в ходе экспериментов в нескольких браузерах, особенность считается достаточно стабильной для начала активного использования. Категорически рекомендуется начинать работу с новыми вариантами сразу же после их выхода, не дожидаясь официального голосования по стандартизации. Если вы еще не знакомы с ES6, считайте, что *время ушло!*

На момент написания книги список предложений и их статус можно было посмотреть здесь: <https://github.com/tc39/ecma262#current-proposals>.

Транскомпиляторы и полизаполнения дают нам способ применять новые функциональные особенности, несмотря на то что они реализованы не во всех браузерах. Несколько основных транскомпиляторов, в том числе Babel и Traceur, уже умеют работать с кое-какими вещами, которые с большой вероятностью станут применяться в следующей за ES6 версии стандарта.

С учетом всего вышесказанного пришло время пристально взглянуть на некоторые из этих функциональных особенностей — чем мы сейчас и займемся!



Все эти функциональные возможности пока находятся на различных стадиях разработки. Велика вероятность, что они начнут использоваться и даже будут иметь такой же вид, как сейчас, тем не менее представленную в последней главе информацию не стоит воспринимать как окончательную. В следующих изданиях этой серии она будет редактироваться с учетом стадий работы над описанными (и над еще не описанными) вариантами новых функциональных особенностей.

Асинхронные функции

В разделе «Генераторы и обещания» главы 4 я упоминал о предложении организовать непосредственную синтаксическую поддержку шаблона для порождающих обещания генераторов в виде средства запуска, которое будет возобновлять работу генератора после выполнения обещания. Посмотрим на предложенную функциональную возможность, которая называется `async function`.

Возьмем пример с генератором из главы 4:

```
run( function *main() {  
  var ret = yield step1();  
  
  try {
```

```
        ret = yield step2( ret );
    }
    catch (err) {
        ret = yield step2Failed( err );
    }

    ret    = yield Promise.all([
        step3a( ret ),
        step3b( ret ),
        step3c( ret )
    ]));

    yield step4( ret );
} )
.then(
    function fulfilled(){
        // '*main()' успешно завершена
    },
    function rejected(reason){
        // Ой, что-то пошло не так
    }
);
```

Предложенный синтаксис `async function` демонстрирует ту же самую логику управления потоком, но без необходимости прибегать к методу `run(...)`, потому что интерпретатор JS будет автоматически знать способы поиска обещаний, находящихся в состоянии ожидания, и возобновлять их работу.

Рассмотрим пример:

```
async function main() {
    var ret = await step1();

    try {
        ret = await step2( ret );
    }
    catch (err) {
        ret = await step2Failed( err );
    }

    ret    = await Promise.all( [
        step3a( ret ),
```

```
        step3b( ret ),
        step3c( ret )
    ] );

    await step4( ret );
}

main()
.then(
    function fulfilled(){
        // 'main()' успешно завершена
    },
    function rejected(reason){
        // Ой, что-то пошло не так
    }
);
```



Также была предложена функциональная особенность `async function*` — асинхронный генератор. Вы сможете использовать в рамках одного кода ключевые слова `yield` и `await` и даже комбинировать их в одном операторе: `x = await yield y`. Эта функциональная особенность пока находится в состоянии обсуждения, в частности, еще не проработана концепция возвращаемого ею значения. Некоторые считают, что оно должно быть *наблюдаемым*, то есть представлять собой своего рода комбинацию между итератором и обещанием. Больше подробностей я вам не расскажу, следите за обновлениями.

Вместо объявления `function *main() { .. }` мы используем форму `async function main() { .. }`, а вместо оператора `yield` с обещанием будет работать оператор `await`. Вызов, запускающий функцию `main()`, на самом деле возвращает обещание, которое мы можем непосредственно наблюдать. Оно эквивалентно тому, что мы получаем от вызова `run(main)`.

Видите симметрию? По сути, `async function` представляет собой синтаксическое сокращение для шаблона использования генераторов с обещаниями и методом `run(..)`; эта функциональная возможность дает точно такой же результат!

Если вы занимаетесь разработкой на языке C# и ключевые слова `async/await` вам уже знакомы, я добавлю, что идея непосредственно навеяна соответствующими операциями из этого языка. Приятно видеть такое взаимопроникновение.

Транскомпиляторы Babel, Traceur и другие уже поддерживают `async function` в ее текущем состоянии, и ничто не мешает вам пользоваться ею. Впрочем, в следующем разделе вы узнаете, почему с этим все-таки пока лучше не торопиться.

Оговорки

Пока нет единства мнений в споре вокруг асинхронной функции. Так как она возвращает только обещание, получается, что снаружи невозможно *отменить* функцию, исполняемую в данный момент. В случае асинхронной операции, требующей большого количества ресурсов, когда вы хотите освободить их сразу же после того, как станет ясно, что ее результат вам не требуется, это становится проблемой.

Рассмотрим пример:

```
async function request(url) {
    var resp = await (
        new Promise( function(resolve,reject){
            var xhr = new XMLHttpRequest();
            xhr.open( "GET", url );
            xhr.onreadystatechange = function(){
                if (xhr.readyState == 4) {
                    if (xhr.status == 200) {
                        resolve( xhr );
                    }
                    else {
                        reject( xhr.statusText );
                    }
                }
            };
            xhr.send();
        } )
    );
};
```

```
    return resp.responseText;
}

var pr = request( "http://some.url.1" );

pr.then(
  function fulfilled(responseText){
    // успешное завершение ajax
  },
  function rejected(reason){
    // Ой, что-то пошло не так
  }
);
```

Предложенный мной в данном фрагменте метод `request(..)` до известной степени напоминает метод `fetch(..)`, который недавно предложили включить в веб-платформу. Интересно, что произойдет, если мы воспользуемся значением `pr`, чтобы каким-то образом обозначить свое желание прервать, например, затянувшийся запрос Ajax?

Обещания не допускают отмены (по крайней мере, не допускали на момент написания этой книги). С моей точки зрения, которую разделяют многие другие разработчики, ее и не нужно допускать (см. книгу *Async & Performance*). Даже если предположить, что у обещания есть метод `cancel()`, означает ли это, что вызов `pr.cancel()` должен распространять сигнал отмены по всей цепочке обещания вплоть до асинхронной функции?

Были предложены несколько вариантов выхода из описанной ситуации:

- асинхронные функции вообще не должны допускать отмены (текущее положение дел);
- в асинхронную функцию во время вызова передается «маркер отмены»;
- возвращаемое значение приводится к типу, допускающему отмену обещаний, который также планируется добавить в стандарт;

- возвращаемое значение меняется на нечто, не являющееся обещанием (например, на наблюдаемое значение или управляющий маркер с обещанием и возможностью отмены).

На момент написания этой книги асинхронные функции возвращали обычные обещания, потому маловероятно, что возвращаемые значения когда-либо будут изменены. Но говорить о том, чем все закончится, пока слишком рано. Следите за новостями и обсуждениями.

Метод `Object.observe(..)`

Одно из сокровенных желаний программистов, занимающихся веб-разработкой на стороне клиента, — это связывание данных, то есть прослушивание обновлений объекта данных и синхронизация их DOM-представлений. Большинство JS-фреймворков предоставляют какие-либо механизмы для таких операций.

Есть большая вероятность, что в следующей за ES6 версии мы увидим поддержку этой функциональной возможности, добавленную непосредственно в язык как метод `Object.observe(..)`. Идея состоит в том, что мы настраиваем обработчик для наблюдения за объектом и при любых изменениях инициируем обратный вызов, после чего соответствующим образом обновляем объектную модель документа.

Мы можем наблюдать за шестью типами изменений:

- `add`;
- `update`;
- `delete`;
- `reconfigure`;
- `setPrototype`;
- `preventExtensions`.

По умолчанию приходят уведомления об изменениях всех типов, но можно отфильтровать только те, которые нам нужны.

Рассмотрим пример:

```
var obj = { a: 1, b: 2 };

Object.observe(
  obj,
  function(changes){
    for (var change of changes) {
      console.log( change );
    }
  },
  [ "add", "update", "delete" ]
);

obj.c = 3;
// { name: "c", object: obj, type: "add" }

obj.a = 42;
// { name: "a", object: obj, type: "update", oldValue: 1 }

delete obj.b;
// { name: "b", object: obj, type: "delete", oldValue: 2 }
```

В дополнение к основным "add", "update" и "delete" меняются следующие типы.

- Событие изменения "reconfigure" возникает, если методом `Object.defineProperty(..)` меняется конфигурация одного из свойств объекта, например редактируется атрибут `writable`. Более подробно эта тема рассматривается в книге *this & Object Prototypes*.
- Событие изменения "preventExtensions" возникает, если при помощи метода `Object.preventExtensions(..)` объект превращают в нерасширяемый.

Так как методы `Object.seal(..)` и `Object.freeze(..)` влекут за собой вызов `Object.preventExtensions(..)`, они также станут активизировать соответствующее событие изменения. Кроме того, события

изменения "reconfigure" будут возникать для каждого свойства объекта. Событие изменения "setPrototype" возникает, когда редактируется [[Prototype]] объекта при помощи метода доступа `__proto__` или посредством метода `Object.setPrototypeOf(..)`.

Обратите внимание, что сведения об этих событиях поступают сразу после изменений. Не путайте их с прокси-объектами (см. главу 7), в случае с которыми действие можно прервать до его возникновения. Наблюдение за объектами позволяет реагировать на уже произошедшие изменения.

Пользовательские события изменений

В дополнение к шести встроенным событиям изменений можно также слушать и создавать пользовательские типы таких событий.

Рассмотрим пример:

```
function observer(changes){
  for (var change of changes) {
    if (change.type == "recalc") {
      change.object.c =
        change.object.oldValue +
        change.object.a +
        change.object.b;
    }
  }
}

function changeObj(a,b) {
  var notifier = Object.getNotifier( obj );

  obj.a = a * 2;
  obj.b = b * 3;

  // поочередно помещаем события изменений в набор
  notifier.notify( {
    type: "recalc",
    name: "c",
    oldValue: obj.c
  } );
}
```

```
}  
var obj = { a: 1, b: 2, c: 3 };  
  
Object.observe(  
  obj,  
  observer,  
  ["recalc"]  
);  
  
changeObj( 3, 11 );  
  
obj.a;           // 12  
obj.b;           // 30  
obj.c;           // 3
```

Набор изменений (пользовательское событие "recalc") поставлен в очередь для доставки обработчику, но пока еще не доставлен, поэтому свойство `obj.c` до сих пор имеет значение 3.

Изменения по умолчанию доставляются в конце текущего цикла событий (см. книгу *Async & Performance*). Для немедленной доставки нужно воспользоваться методом `Object.deliverChangeRecords(observer)`. Сделав это, вы увидите, как ожидаемым образом изменится значение свойства:

```
obj.c;           // 42
```

В предыдущем примере мы вызывали метод `notifier.notify(..)` с полной записью события изменения. Альтернативный способ создания очереди из записей изменений — использование метода `performChange(..)`, который отделяет записи о событии от остальных свойств, указывая тип события (через обратный вызов функции). Рассмотрим пример:

```
notifier.performChange( "recalc", function(){  
  return {  
    name: "c",  
    // 'this' является наблюдаемым объектом  
    oldValue: this.c  
  };  
} );
```

В определенных обстоятельствах такое разделение функциональных возможностей способно более аккуратно проецироваться на ваш шаблон использования.

Завершение наблюдения

Как и в случае с обычными обработчиками событий, может возникнуть ситуация, когда вы хотите прекратить наблюдение за событиями изменения объекта. Это делается через метод `Object.unobserve(...)`.

Рассмотрим пример:

```
var obj = { a: 1, b: 2 };

Object.observe( obj, function observer(changes) {
    for (var change of changes) {
        if (change.type == "setPrototype") {
            Object.unobserve(
                change.object, observer
            );
            break;
        }
    }
} );
```

В этом простом примере мы слушаем события изменений, пока не возникает событие "setPrototype", после чего наблюдение прекращается.

Оператор возведения в степень

Для JavaScript был предложен оператор, выполняющий возведение в степень таким же способом, как это делает метод `Math.pow(..)`. Рассмотрим пример:

```
var a = 2;
```

```
a ** 4;           // Math.pow( a, 4 ) == 16
```

```
a **= 3;           // a = Math.pow( a, 3 )
a;                // 8
```



Оператор ******, по сути, — тот же самый, что и в языках Python, Ruby, Perl и др.

Свойства объектов и оператор

Как было показано в разделе «Слишком много, слишком мало, в самый раз» главы 2, оператор **...** функционирует совершенно очевидным образом, разбивая или собирая массивы. А как он поступает с объектами?

Сначала предполагалось, что эта функциональная возможность появится в стандарте ES6, но ее решили отложить до следующей версии (известной как ES7 или ES2016). Вот как она может быть реализована в перспективе:

```
var    o1 = { a: 1, b: 2 },
      o2 = { c: 3 },
      o3 = { ...o1, ...o2, d: 4 };
console.log( o3.a, o3.b, o3.c, o3.d );
// 1 2 3 4
```

Также оператор **...** можно применять для сбора деструктурированных свойств объекта обратно в объект:

```
var o1 = { b: 2, c: 3, d: 4 };
var { b, ...o2 } = o1;

console.log( b, o2.c, o2.d );           // 2 3 4
```

В данном фрагменте кода оператор **...o2** собирает деструктурированные свойства **c** и **d** обратно в объект **o2** (у него отсутствует свойство **b**, которое есть у объекта **o1**).

Еще раз напоминаю, что это всего лишь предложение, и оно будет рассматриваться только после завершения работы над ES6. Впрочем, если бы это реализовали — вышло бы здорово.

Метод `Array#includes(..)`

Поиск в массиве значений — это задача, с которой постоянно приходится иметь дело разработчикам JS. Обычно она решалась следующим образом:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.indexOf( 42 ) >= 0) {
    // найдено!
}
```

Проверка `>= 0` выполняется потому, что метод `indexOf(..)` при успешном поиске возвращает численное значение 0 или выше, а в случае неудачи — 1. Другими словами, мы используем функцию, возвращающую индексы в булевом контексте. Но так как значение `-1` оценивается как истинное, а не как ложное, проверка требует от нас дополнительных действий.

В книге *Types & Grammar* этой серии я рассмотрел другой вариант, который мне нравится несколько больше:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (~vals.indexOf( 42 )) {
    // найдено!
}
```

В данном случае оператор `~` сопоставляет значение, возвращаемое методом `indexOf(..)`, с диапазоном значений, приводимых к типу `boolean`. В результате `-1` даст нам `0` (ложь), а все остальное — ненулевые значения (истина), что нам, собственно, и требуется, чтобы понять, обнаружено значение или нет.

Я считаю, что это шаг вперед, но многие категорически со мной не согласны. Впрочем, никто не может доказать идеальность логической схемы поиска в методе `indexOf(..)`. Например, она не позволяет искать в массивах значения `NaN`.

В результате появилось и получило активную поддержку предложение добавить метод поиска в массивах, возвращающий логическое значение, которое называется `includes(..)`:

```
var vals = [ "foo", "bar", 42, "baz" ];

if (vals.includes( 42 )) {
    // найдено!
}
```



Метод `Array#includes(..)` использует алгоритм поиска, который обнаруживает значения `NaN`, но не делает различий между значениями `-0` и `0` (см. книгу *Types & Grammar* этой серии). Если для вашей программы значения `-0` — не проблема, значит, вы нашли оптимальный вариант. Если же вы о них думаете, то вам придется создать собственный поисковый алгоритм, например, на основе метода `Object.is(..)` (см. главу 6).

Принцип SIMD

Детально принцип «одиночный поток команд, множественный поток данных» (SIMD — single instruction, multiple data) рассматривается в книге *Async & Performance*, но здесь его тоже стоит упомянуть, так как эта функциональная особенность, вероятно, появится в будущих версиях языка JS.

SIMD API предлагает различные низкоуровневые инструкции (для CPU), позволяющие манипулировать более чем одним численным значением за один раз. Например, можно определить два вектора из четырех и из восьми чисел и мгновенно перемножить соответствующие элементы (параллелизм данных).

Рассмотрим пример:

```
var v1 = SIMD.float32x4( 3.14159, 21.0, 32.3, 55.55 );
var v2 = SIMD.float32x4( 2.1, 3.2, 4.3, 5.4 );

SIMD.float32x4.mul( v1, v2 );
// [ 6.597339, 67.2, 138.89, 299.97 ]
```

Принцип SIMD включает в себя и ряд других операций, кроме `mul(..)` (умножение), в том числе `sub()`, `div()`, `abs()`, `neg()`, `sqrt()` и многие другие.

Параллельные математические операции имеют большое значение для следующего поколения высокопроизводительных приложений JS.

Язык WebAssembly (WASM)

Когда работа над первым изданием этой книги почти подошла к концу, Брендан Эйх сделал заявление, которое, вероятно, окажет значительное влияние на будущее JavaScript: он сказал о WebAssembly (WASM). Детально осветить этот язык невозможно, так как на момент написания книги говорить о чем-то подобном было рано. Но без хотя бы краткого его упоминания книга кажется мне неполной.

На недавние проектные изменения языка сильнее всего воздействовало желание сделать его более подходящим для транскомпиляции/кросскомпиляции из других языков (например, C/C++, ClojureScript и т. п.). Очевидно, что в первую очередь приходится заботиться о производительности кода, запускаемого как JavaScript.

В книге *Async & Performance* этой серии рассказывается, что несколько лет назад группа разработчиков из компании Mozilla выдвинула идею подмножества для JavaScript, которое называлось ASM.js. Оно значительно ограничивало определенные действия,

затрудняющие оптимизацию кода для движка JS. В результате умеющий работать с этим подмножеством движков обрабатывал совместимый с ASM.js код значительно быстрее, практически с той же скоростью, что и оптимизированные эквиваленты на языке C. Многие рассматривают ASM.js как наиболее вероятную основу, на которой будут работать с JavaScript приложения, требующие больших ресурсов производительности.

Другими словами, все дороги к запуску кода в браузерах *идут через JavaScript*.

Вернее, так было до появления WASM. Он дает остальным языкам альтернативный способ обращаться к среде исполнения браузера, не проходя перед этим через JavaScript. По сути, если WASM начнет широко использоваться, движки JS получат дополнительную способность выполнять код в бинарном формате, в некотором отношении рассматриваемый как байт-код (напоминающий тот, что запускается на JVM).

Язык WASM предлагает формат для бинарного представления сильно сжатого AST (синтаксического дерева) кода, позволяющий передавать инструкции непосредственно движку JS без необходимости предварительного прогона через интерпретатор JS или даже ведущий себя по правилам JS. Такие языки, как C или C++, позволяют компиляцию непосредственно в формат WASM вместо ASM.js и получают преимущество в виде дополнительной скорости работы за счет пропуска этапа анализа JS.

Ближайшая цель языка WASM — достижение паритета с подмножеством ASM.js и далее с языком JS. В конечном счете ожидается, что WASM даст новые возможности, превосходящие все, на что способен JS. Например, в JS присутствует спрос на развитие фундаментальных функциональных особенностей, таких как потоки, — это изменение, несомненно, сильно потрясет всю экосистему JavaScript. Но куда более многообещающей выглядит реализация этой функциональной возможности в будущем расширении WASM, избавляющем от необходимости вносить коренные изменения в JS.

По большому счету, такой сценарий открывает новые пути к использованию различных языков программирования во Всемирной паутине. Это потрясающая возможность развития веб-платформы!

Что все вышесказанное означает для JS? Он устареет или умрет? К этому нет никаких предпосылок. С большой вероятностью года через два выйдет из употребления подмножество ASM.js, но основная часть языка JS слишком сильно связана с веб-платформой.

Сторонники WASM предполагают, что успех этого языка будет означать защиту JS от чрезмерного спроса на функциональные особенности, способные в конечном счете увести процесс расширения языка за грань разумного. По прогнозам, именно WASM станет предпочтительной целью для высокопроизводительных частей приложений.

Что интересно, JavaScript — это один из языков, которые, скорее всего, в будущем не станут ориентироваться на WASM. Конечно, возможны изменения, способные сформировать подходящие для такого ориентирования подмножества JS, но этот путь развития не выглядит перспективным.

Несмотря на то что JS, скорее всего, не станет сосредотачиваться на WASM, код, написанный на этих двух языках, сможет серьезно взаимодействовать, причем таким же естественным образом, как в случае взаимодействия модулей. Представьте, как вызов функции JS `foo()` приводит к активизации функции WASM с таким же именем, способной работать, не сталкиваясь с ограничениями остального кода на JS.

Все то, что сейчас пишут на JS, вероятно, и далее будут писать на этом языке, по меньшей мере, в обозримой перспективе. А вот вещи, транскомпилируемые на JavaScript, скорее всего, будут переориентированы на WASM. Для приложений, которым требуется максимальная производительность с минимальной чувствительностью к слоям абстракции, наилучшим вариантом станет поиск подходящего языка, отличного от JS, и последующее переориентирование на WASM.

С большой вероятностью эти изменения будут происходить медленно, в течение многих лет. Основные браузерные платформы начнут поддерживать WASM в лучшем случае через несколько лет. Пока же для проекта WASM (<https://github.com/WebAssembly>) существует полизаполнение, призванное продемонстрировать работоспособность его основных принципов.

Можно предположить, что по мере того как в WASM начнут появляться новые, не связанные с JS приемы, некоторые вещи, в настоящее время типичные для JS, будут переписываться на языке, ориентированном на WASM. Таким образом получают преимущества, например, те части фреймворков, для которых важна производительность, игровые движки и остальные интенсивно используемые инструменты.

Программисты, работающие с этими инструментами в своих веб-приложениях, скорее всего, не заметят особых изменений, а просто автоматически начнут пользоваться преимуществами производительности и новыми функциональными особенностями.

Уверенно можно сказать только одно. Чем более реальным становится язык WASM с течением времени, тем больше он влияет на путь развития и проектирования JavaScript. Пожалуй, это одна из наиболее важных тем, которые будут рассматриваться в версиях стандарта после ES6, так что разработчикам следует внимательно за ней следить.

Подводим итоги

Если все остальные книги этой серии были написаны в ключе «вы, возможно, знаете JS не настолько хорошо, как думаете», то здесь вопрос стоит так: «вы больше не знаете JS». В книге рассматривается множество новых вещей, добавленных в язык спецификацией ES6. Это восхитительное собрание функциональных особенностей и парадигм, которые сделают ваши программы лучше.

Однако развитие языка JS не останавливается на версии ES6! Об этом нельзя даже думать. Есть новые функциональные особенности, которые войдут в следующие версии и уже находятся на разных стадиях разработки. В этой главе мы кратко рассмотрели наиболее вероятных кандидатов на скорейшее воплощение.

Асинхронные функции представляют собой мощный инструмент, сокращающий шаблон «генераторы и обещания» (см. главу 4). Метод `Object.observe(...)` позволяет непосредственно наблюдать за изменениями объектов, что крайне важно для реализации связывания данных. Оператор возведения в степень `**`, оператор `...` для свойств объектов и метод `Array#includes(...)` — все это простые, но крайне полезные усовершенствования уже существующих механизмов. Наконец, принцип SIMD возвещает новую эру в развитии высокопроизводительного JS.

Как бы избито ни звучало, но JS в самом деле ждет светлое будущее! Ответы на вопросы, поставленные в книгах серии *You Don't Know JS* и, в частности, в этой, теперь предстоит искать читателям. Чего вы ждете? Пришло время исследования и экспериментов!

К. Симпсон
ES6 и не только
Перевела с английского И. Рузмайкина

Заведующая редакцией	<i>Ю. Сергиенко</i>
Ведущий редактор	<i>Н. Римицан</i>
Литературный редактор	<i>А. Петров</i>
Художник	<i>С. Заматевская</i>
Корректор	<i>С. Беляева</i>
Верстка	<i>Л. Соловьева</i>

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —
Книги печатные профессиональные, технические и научные.

Подписано в печать 22.07.16. Формат 60×90/16. Бумага офсетная. Усл. п. л. 21,000.
Тираж 1000. Заказ 0000.

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

Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87