КАЙЛ СИМПСОН

ПОЗНАКОМЬТЕСЬ, JAVASCRIPT

2-Е ИЗДАНИЕ

ВЫ ПОКА ЕЩЕ НЕ ЗНАЕТЕ JS

You Don't Know JS Yet: Get Started

Get to know JS

Kyle Simpson

КАЙЛ
СИМПСОН

ПОЗНАКОМЬТЕСЬ, JAVASCRIPT

2-Е МЕЖДУНАРОДНОЕ ИЗДАНИЕ

{ ВЫ ПОКА ЕЩЕ НЕ ЗНАЕТЕ JS }

ПИТЕР®

Санкт-Петербург • Москва • Минск

2022

ББК 32.988.02-018
УДК 004.738.5
С37

Симпсон Кайл

С37 {Вы пока еще не знаете JS} Познакомьтесь, JavaScript. 2-е изд. — СПб.: Питер, 2022. — 192 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1875-5

Вы пока еще не знаете JS. И Кайл Симпсон признается, что тоже его не знает (по крайней мере полностью)… И никто не знает. Но все мы можем начать работать над тем, чтобы узнать его лучше. Сколько бы времени вы ни провели за изучением языка, всегда можно найти что-то еще, что стоит изучить и понять на другом уровне.Учтите, что, хотя книга и называется «Познакомьтесь, JavaScript», она не для новичков. У нее другая задача: дать обзор тем, в которых необходимо разобраться на начальном этапе изучения JS. Даже если вы уже написали достаточно кода JS, эту книгу не стоит пропускать, возможно, в ваших знаниях есть пробелы, которые необходимо заполнить перед углубленным изучением сложных тем. Пора начать изучение JS!

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

ББК 32.988.02-018
УДК 004.738.5

ISBN 979-8602477429 англ.
ISBN 978-5-4461-1875-5

Оглавление

Благодарности

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

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

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

Предисловие

Когда я впервые увидел твит с рекламой сбора денег на оригинальную серию книг «Вы не знаете JS», я подумал: кем бы ни был этот Кайл Симпсон, пускай идет лесом. Конечно же, я знаю JavaScript! В то время я работал на JavaScript уже много лет с самыми авторитетными представителями отрасли и поэтому считал, что на подобные обобщения могу смотреть свысока.

После того как кампания завершилась, я заметил, что многие люди с большим энтузиазмом относятся к этой серии. И тогда я решил все же попробовать — просто чтобы показать всем, что я знаю JavaScript. Но когда я погрузился в материал и стал внимательно изучать текст, то испытал глубокое чувство удивления, любопытства и даже некоторого раздражения. У Кайла настоящий дар говорить что-то такое, что противоречит моим представлениям о мире, и заставлять меня думать об этом, пока я не пойму, что сказанное было правдой (хотя я ему в этом никогда не признаюсь).

В общем, выяснилось, что JavaScript я не знал. Я не знал, почему были приняты те или иные паттерны; я не знаю, почему в некоторых ситуациях JavaScript ведет себя именно так, а не иначе; я не знал многие нюансы языка, которые, как мне казалось, были мне известны. Я в принципе не догадывался, что не знал многих аспектов, и это снижало мою квалификацию как разработчика.

Именно этим и примечательна книга. Она написана не только для тех, кто хочет освоить новый язык (хотя и для них тоже); она написана для ремесленников от программирования, которые хотят мастерски овладеть своими инструментами, понимать все тонкости своего ремесла и выбирать подходящие средства для решения задач.

В Кайле и его работе мне прежде всего нравится то, что он остается восхитительно непоколебимым перед влиянием окружающего его мира программирования. Это вовсе не означает, что он не в курсе происходящего в сообществе; скорее он не отклоняется от своего поиска лучших ответов на правильно поставленные вопросы. Часто он начинает противоречить новейшим «лучшим практикам», но в действительности это именно то, что нужно: точка зрения, которая вне времени и просто выявляет истину. Этим-то так хороша эта серия. Первое издание «Вы не знаете JS» остается актуальным даже годы спустя! Немногие

книги выдержали проверку временем с учетом меняющегося ландшафта JavaScript.

Несколько слов о первой книге, «Познакомьтесь, JavaScript». Возможно, у вас возникнет соблазн пропустить ее, потому что вы думаете, что «первые шаги» уже давно сделаны. И все же вам стоит ее прочитать! Вы не поверите, сколько глубины, нюансов и странностей кроется в основных структурных элементах JavaScript. Очень важно, чтобы вы освоили всю эту подоплеку, прежде чем переключаться на конструкции языка. Именно такой фундамент понадобится вам для того, чтобы действительно хорошо знать JavaScript.

Итак, окажите себе услугу, основательно прочитайте эту книгу и высвободите содержащиеся в ней знания. Этот надежный фундамент послужит вам лучше, чем любой фреймворк или библиотека; они приходят и уходят, но все мы еще будем писать код JavaScript в ближайшие десятилетия. Сохраняйте объективность и ставьте под сомнение свои стереотипы.

Потому что, как я сам убедился, скорее всего, вы не знаете JavaScript (пока).

Брайан Холт (Brian Holt),
старший разработчик Visual Studio Code
и Node.js для Azure Microsoft

Вступление

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

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

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

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

Если вы недавно занимаетесь программированием или JS, то учтите, что эти книги не задумывались как «деликатный вводный курс по JavaScript». Временами материал становится сложным и требующим се-

рьезных усилий для понимания, и многие темы рассматриваются намного глубже, чем в книгах для новичков. Книга может пригодиться всем читателям независимо от уровня подготовки, но я писал ее с прицелом на то, что вы уже знакомы с JS, а ваш практический опыт работы с этим языком составляет хотя бы полгода, если не больше.

Части языка

В этих книгах я намеренно отошел от традиционного подхода, в котором рассматриваются *хорошие части* языка. Нет, это не означает, что мы будем рассматривать только *плохие части* — скорее рассматриваться будут **все части**.

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

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

Как и у любого великого языка, у него есть как выдающиеся достоинства, так и недостатки. Даже сам создатель JavaScript Брендан Эйх сожалеет по пово-

ду некоторых частей и называет их ошибками. Но он заблуждается: они вовсе не были ошибками. В наши дни JS стал тем, чем он стал — самым распространенным, а следовательно, самым влиятельным языком программирования, — именно из-за *всех этих частей*.

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

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

На мой взгляд, абсурдно даже пытаться стать по-настоящему эффективным разработчиком JavaScript, используя только узкий срез возможностей языка. Можно ли представить рабочего с полным ящиком инструментов, который пользуется только молотком, а отвертку и рулетку презирает, считая их недостойными? Это просто глупо.

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

Название?

Какой же смысл заложен в название серии?

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

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

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

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

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

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

Рекомендую сформировать особую систему взглядов на JavaScript (и на разработку в целом): вы никогда не освоите его полностью, но можете (и должны) работать над тем, чтобы приблизиться к этой цели.

Этот путь растянется на всю вашу карьеру разработчика и даже дальше.

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

Миссия

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

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

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

Мой опыт не только как разработчика, но и как преподавателя говорит мне: вы всегда можете повысить эффективность своего труда, если четко будете понимать, как работает ваш код (а не просто добиваться того, чтобы он выдавал желаемый результат).

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

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

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

Путь

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

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

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

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

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

Моя цель не навязать вам свое мнение. Скорее я хочу выработать у вас собственное мнение и умение его отстаивать. Вы не сможете достичь *этой цели* скоростным чтением. На это уйдет немало времени. Вы будете двигаться вперед шаг за шагом, пока изучаете, размышляете и возвращаетесь к прочитанному. Эти книги были задуманы как путеводители по JavaScript от вашего текущего местонахождения в знаниях о язы-

ке до точки более глубокого понимания. А теперь самая интересная часть: чем глубже вы понимаете JS, тем больше вопросов у вас появится и тем больше придется изучать!

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

От издательства

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

Мы будем рады узнать ваше мнение!

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

1 Что такое JavaScript?

Вы пока еще не знаете JS. Я его тоже не знаю (по крайней мере, полностью)... И никто не знает. Но все мы можем хотя бы начать работать над тем, чтобы узнать его лучше.

В главе 1 мы постараемся заложить основу для дальнейшего движения вперед. Мы начнем с рассмотрения разных малозаметных, но важных подробностей, а заодно развеем некоторые мифы и ошибочные представления о том, чем в действительности является язык (и чем он не является!).

Это исключительно ценная информация о том, как устроен язык JS и как он развивается. Если вы решили лучше узнать JS, то путешествие следует начинать именно с этих первых шагов.

О книге

Обратите особое внимание на слово «путешествие», потому что хорошее знание JS — не пункт назначения, а направление. Сколько бы времени вы ни провели за изучением языка, всегда можно найти что-то еще. В общем, не рассчитывайте, что вы сможете галопом промчаться по страницам книги и быстро получить результат. Пока вы будете делать эти первые шаги, вашими лучшими друзьями будут терпение и настойчивость.

После этой вводной главы в оставшейся часть книги на высоком уровне будет представлено то, что вы узнаете, когда будете разбираться в JS и изучать его в книгах YDKJSY.

В частности, в главе 4 представлены три столпа, на которых строится язык JS: области видимости/замыкания, прототипы/объекты и типы/преобразования. JS — обширный и нетривиальный язык со множеством возможностей и инструментов. При этом все в нем основано на этих трех столпах.

Учтите, что хотя книга и называется «Познакомьтесь, JavaScript», она не для новичков. Главная задача книги — подготовить читателя к глубокому изучению JS по остальным книгам серии; я писал ее, предполагая, что вы работали с JS хотя бы несколько месяцев, прежде чем взяться за эту серию. Итак, чтобы книга принесла максимальную пользу, вы должны потратить немало времени на написание кода JS и накопить практический опыт.

Даже если опыт у вас солидный, эту книгу не стоит пролистывать или что-то пропускать. Не жалейте времени на полное усвоение материала. Хорошее начало всегда зависит от уверенных первых шагов.

Откуда взялось название?

Пожалуй, JavaScript — самое ошибочное и неправильно понимаемое название языка программирования.

Связан ли этот язык с Java? Он является сценарной формой Java? Он предназначен для написания сценариев, а не настоящих программ?

Дело в том, что название JavaScript появилось из-за маркетинговых ухищрений. Когда Брендан Эйк впервые задумывал этот язык, он присвоил ему условное наименование Mocha. Во внутренних коммуникациях Netscape использовалось название LiveScript. Но когда наступило время выбрать название «для общественности», победило название JavaScript.

Почему? Да потому, что язык изначально проектировался для аудитории, состоящей в основном из программистов Java, а слово script («сценарий») в то время часто использовалось для обозначения упрощенных программ. Этим упрощенным «сценариям» было суждено стать первым кодом, встраиваемым в страницы модной новинки — Всемирной паутины!

Другими словами, название JavaScript было маркетинговой уловкой, которая пыталась выдать этот язык за возможную альтернативу для более тяжеловесного и более известного языка Java. Если уж на то пошло, с таким же успехом можно было назвать язык Web-Java.

Между кодом Java и JavaScript существует некоторое поверхностное сходство. Эти схожие черты появились не столько из-за общего происхождения, сколько из-за того, что оба языка ориентировались на разработчиков, привычных к синтаксису C (и в определенной степени C++).

Например, символ `{` открывает блок кода, а символ `}` закрывает его, как и в C/C++ или Java. Конец команды также отмечается символом `;`.

В определенном отношении родство уходит глубже синтаксиса. Oracle — компании, которая все еще является владелицей языка Java и распоряжается им (после Sun), — также принадлежит официальный товарный знак JavaScript (после Netscape). Этот товарный знак почти никогда не становится предметом юридической защиты прав по его использованию, и на данный момент это маловероятно.

По этим причинам некоторые читатели предложили использовать JS вместо JavaScript. Это очень распространенное сокращение и, пожалуй, хороший кандидат для официального переименования самого языка. В этих книгах будет почти повсеместно использоваться сокращение JS.

Есть еще одно обстоятельство, которое еще сильнее отдаляет язык от принадлежащего Oracle товарного знака. Дело в том, что официально язык называется ECMAScript; это название определено TC39 и фор-

мализовано комитетом по стандартизации ECMA. А с 2016 года к официальному названию языка также присоединяется год ревизии; на момент написания книги это ECMAScript 2019, или сокращенно ES2019.

Иначе говоря, JavaScript/JS, выполняемый в вашем браузере или в Node.js, является *реализацией* стандарта ES2019.

Не используйте для обозначения языка такие термины, как JS6 или ES8. Некоторые авторы это делают, но лишние термины только усугубляют путаницу. Придерживайтесь названий ES20xx или просто JS.

Но как бы вы ни называли язык — JavaScript, JS, ECMAScript или ES2019, он, безусловно, не является разновидностью языка Java!

> У Java с JavaScript столько же общего, как у Луны с луна-парком.
>
> *Джереми Кит (Jeremy Keith), 2009*

Спецификация языка

Я уже упоминал: TC39 — технический координационный комитет, управляющий JS. Главной задачей комитета является создание официальной специфи-

кации языка. Участники регулярно встречаются для голосований по согласованным изменениям, которые затем передаются ECMA (комитету по стандартизации).

Синтаксис и поведение JS определяются в спецификации ES.

Так уж получилось, что ES2019 является 10-й основной спецификацией/ревизией с момента принятия JS в 1995 году, так что в официальный URL-адрес спецификации, размещенной ECMA, входит часть «10.0»:

https://www.ecma-international.org/ecma-262/10.0/

В комитет TC39 входят от 50 до 100 представителей компаний, занимающихся веб-технологиями, включая разработчиков браузеров (Mozilla, Google, Apple) и устройств (Samsung и т. д.). Все участники комитета являются волонтерами, хотя многие из них работают в этих компаниях и могут получать частичную компенсацию за свои обязанности в комитете.

Встречи TC39 обычно проводятся каждые два месяца и занимают около 3 дней. На них участники отчитываются о работе, выполненной с момента последней встречи, обсуждают возникшие проблемы и голосуют по предложениям. Места проведения встреч чередуются между компаниями-участницами.

Все предложения TC39 проходят процесс, состоящий из 5 этапов — с 0 по 4 (а как иначе, мы же программисты!). Дополнительную информацию о процессе можно найти по адресу https://tc39.es/process-document/.

Этап 0 означает примерно следующее: кто-то из участников TC39 считает, что идея стоящая, планирует взяться за нее и поработать над ней. Таким образом, многие идеи, «предлагаемые» людьми, не входящими в TC39, по неформальным каналам (социальные сети, публикации в блогах), в действительности находятся в фазе «до этапа 0». Чтобы предложение могло официально считаться относящимся к «этапу 0», за него должен выступить кто-то из участников TC39.

После того как предложение достигнет статуса «этапа 4», оно считается пригодным для включения в ревизию языка в следующем году. На прохождение всех этапов предложению может потребоваться от нескольких месяцев до нескольких лет.

Управление всеми предложениями осуществляется открыто в репозитории Github в TC39: https://github.com/tc39/proposals.

Любой желающий, состоящий в TC39 или нет, может участвовать в публичных обсуждениях и процессах работы над предложениями. Но только участники TC39 могут посещать встречи и голосовать за предложения и изменения. Таким образом, голос участни-

ка TC39 имеет очень большой вес в отношении того, в каком направлении будет двигаться JS. В отличие от распространенного (и, как ни печально, укоренившегося) мифа, никаких множественных версий JS, используемых на практике, *нет*. Есть только одна версия JS — официальный стандарт, находящийся в ведении TC39 и ECMA.

В начале 2000-х, когда компания Microsoft поддерживала JScript — ответвление JS, полученное посредством реверс-инжиниринга (и не обладавшее полной совместимостью), можно было со всем основанием говорить о нескольких версиях JS. Но эти времена давно прошли. В наши дни подобные заявления о JS говорят об отсталости и неточности.

Все крупные производители браузеров и устройств обязались поддерживать соответствие своих реализаций JS этой центральной спецификации. Конечно, в ядрах разные функции реализуются в разное время. При этом никогда не может возникнуть ситуация, когда ядро v8 (ядро JS для Chrome) реализует некоторую возможность иначе или несовместимым образом по сравнению с ядром SpiderMonkey (ядро JS для Mozilla).

Это означает, что вы можете изучить **один вариант JS** и полагаться на него повсеместно.

Веб (JS)

Хотя набор сред, в которых выполняется JS, постоянно расширяется (от браузеров до серверов (Node.js), роботов, умных лампочек...), единственной средой, которая по-настоящему правит JS, является веб.

Иначе говоря, во всех практических отношениях реализация JS для веб-браузеров — единственная реальность, которая действительно имеет значение.

По большей части JS, определенный в спецификации, и JS, работающий в браузерных ядрах JS, совпадают. Однако существуют и некоторые отличия, о которых необходимо помнить.

Иногда спецификация JS диктует новое или уточненное поведение, которое не полностью совпадает с тем, как работают браузерные ядра JS. Такое несоответствие существует исторически: у ядер JS накопилось уже более 20+ лет наблюдаемого поведения граничных случаев функциональности, от которой зависит веб-контент. Как следствие, ядра JS иногда отказываются соответствовать изменениям, продиктованным в спецификации, потому что это может привести к нарушению работы веб-контента.

В таких случаях TC39 часто отступает и просто решает согласовать спецификацию с веб-реальностью. Например, TC39 планировал добавить метод `contains(..)` для `Array`, но выяснилось, что это имя конфликтует

со старыми фреймворками JS, все еще использующимися на некоторых сайтах, поэтому они изменили имя на неконфликтное `includes(..)`. То же произошло с трагикомическим кризисом в сообществе JS, который был прозван smoosh-гейтом, когда запланированный метод `flatten(..)` в конечном итоге получил название `flat(..)`.

Но время от времени TC39 решает, что по некоторым позициям спецификация должна строго соблюдаться, несмотря на то что браузерные ядра JS вряд ли когда-либо будут ей соответствовать.

Решение? Приложение B, Additional ECMAScript Features for Web Browsers[1]. Спецификация JS включает это приложение для описания любых несоответствий между официальной спецификацией JS и реальностью JS в веб. Иначе говоря, есть исключения, разрешенные *только* для веб-JS; все остальные среды JS должны придерживаться буквы закона.

В разделах B.1 и B.2 рассматриваются *дополнения* к JS (синтаксис и API), включенные в веб-JS (снова по историческим причинам), но которые TC39 не планирует формально включать в основную функциональность JS. Примеры — восьмеричные литералы

[1] ECMAScript 2019 Language Specification, Appendix B: Additional ECMAScript Features for Web Browsers, https://www.ecma-international.org/ecma-262/10.0/#sec-additional-ecmascript-features-for-web-browsers (последняя версия на момент написания книги в январе 2020 года).

с префиксом 0, глобальные функции `escape(..)`/`unescape(..)`, вспомогательные методы `String` вроде `anchor(..)` и `blink(..)`, метод `RegExp compile(..)`.

В разделе B.3 описаны некоторые конфликтные ситуации, в которых код может выполняться как в ядрах веб-JS, так и в других ядрах, но в которых *может* наблюдаться разное поведение, приводящее к разным результатам. Большинство перечисленных изменений связано с ситуациями, которые помечаются как ранние ошибки при выполнении кода в строгом (strict) режиме.

Подводные камни из приложения B встречаются нечасто, и все же лучше избегать этих конструкций, чтобы не создавать себе проблем в будущем. По возможности придерживайтесь спецификации JS и старайтесь не зависеть от поведения, применимого только в средах отдельных ядер JS.

Не только (веб) JS...

Является ли следующий код программой JS?

```
alert("Hello, JS!");
```

Все зависит от точки зрения. Функция `alert(..)` не входит в спецификацию JS, но *присутствует* во всех веб-JS средах. Но вы не найдете ее в приложении B. В чем же дело?

Разные среды JS (браузерные ядра JS, Node.js и т. д.) добавляют в глобальную область видимости ваших программ JS различные API, которые представляют функциональность, зависящую от среды, — например, возможность вывести диалоговое окно уведомления в браузере пользователя.

Как выясняется, широкий спектр API, внешне похожих на JS, таких как `fetch(..)`, `getCurrentLocation(..)` и `getUserMedia(..)`, составляют веб-API. В Node.js можно обращаться к сотням методов API из различных встроенных модулей, таких как `fs.write(..)`.

Другой распространенный пример — `console.log(..)` (и все остальные методы `console.*`). В спецификации JS они не указаны, но благодаря своей универсальной полезности они определяются практически во всех средах JS в соответствии с приблизительно достигнутым консенсусом.

Итак, `alert(..)` и `console.log(..)` не определяются в JS, но они *выглядят* как JS. Это функции и методы объектов, подчиняющиеся правилам синтаксиса JS. Лежащем за ними поведением управляет среда, в которой выполняется ядро JS. Однако на поверхности они должны подчиняться правилам JS, чтобы иметь возможность играть на площадке JS.

Многие межбраузерные различия, на которые люди жалуются со словами: «JS такой непоследователь-

ный!», на самом деле обусловлены различиями в поведении сред, а не самого JS.

Итак, вызов `alert(..)` *принадлежит* JS, но сама функция `alert` — всего лишь гость, а не часть официальной спецификации JS.

Не всегда JS

Консоль/REPL (Read-Evaluate-Print-Loop) в средствах разработчика вашего браузера (или Node) на первый взгляд выглядит достаточно элементарно. Но на самом деле впечатление обманчиво.

Средства разработчика... это всего лишь средства разработчика. Они предназначены для того, чтобы упростить их жизнь. На первое место разработчики ставят свой опыт (DX, Developer Experience). Средства разработчика не пытаются точно и однозначно отразить все нюансы поведения JS, строго соответствующие спецификации JS. Из-за этого появляется множество особенностей, которые могут стать «подводными камнями», если относиться к консоли как к *чистой* среде JS.

Кстати говоря, это хорошо! Я рад, что средства разработчика упрощают жизнь разработчиков! Я рад, что у нас есть такие приятные вещи, как автозаполнение переменных/свойств и т. д. Я всего лишь указываю, что мы не можем (и не должны) ожидать, что эти

средства будут всегда строго соответствовать правилам выполнения JS, потому что они создавались не для этого. Так как поведение этих средств изменяется от браузера к браузеру и поскольку они изменяются (иногда достаточно часто), я не собираюсь жестко закреплять никакие конкретные подробности в тексте; это гарантировало бы, что текст книги быстро устареет.

Но я укажу на некоторые примеры аномалий, которые встречались в различных точках разных консольных сред JS. Они помогут мне подкрепить утверждение о том, что при их использовании не следует заранее предполагать платформенное поведение JS.

- Создает ли объявление `var` или `function` в «глобальной области видимости» верхнего уровня консоли полноценную глобальную переменную (а также зеркальное свойство `window`, и наоборот).
- Что происходит с множественными объявлениями `let` и `const` в «глобальной области видимости» верхнего уровня.
- Включает ли команда `"use strict";` в однострочном вводе (с последующим нажатием клавиши Enter) строгий режим на оставшуюся часть консольного сеанса, как бы она сделала в первой строке файла .js, и можно ли включить строгий режим в сеансе при размещении `"use strict";` после «первой строки».

- Как привязка по умолчанию `this` в нестрогом режиме работает с вызовами функций и будет ли использованный «глобальный объект» содержать ожидаемые глобальные переменные.
- Как поднятие (hoisting) (см. книгу 2, «Области видимости и замыкания») работает с многострочными элементами.
- ...И другие примеры.

Консоль разработчика не пытается притворяться компилятором JS, который обрабатывает введенный код в точности по тем же правилам, что и ядро JS при обработке файла .js. Он старается упростить вашу задачу — быстро ввести несколько строк кода и немедленно получить результат. Это совершенно разные сценарии использования, а следовательно, было бы неразумно ожидать, что одна программа будет одинаково вести себя в обоих случаях.

Не думайте, что поведение, которое вы наблюдаете в консоли разработчика, в точности воспроизводит семантику JS; если вас интересует эта тема, читайте спецификацию. Вместо этого лучше рассматривать консоль как JS-совместимую среду. Это может быть полезно само по себе.

Многоликий язык

Термином «парадигма» в контексте языков программирования обозначаются широкий (почти универсальный) менталитет и подход к структурированию кода. В рамках парадигмы существует великое множество разновидностей стиля и формы, по которым программы отличаются друг от друга, включая бесчисленные библиотеки и фреймворки, оставляющие свой неповторимый след на любом коде.

Но каким бы ни был индивидуальный стиль программы, высокоуровневые категории почти всегда очевидны с первого взгляда на любую программу.

В типичной схеме классификации кода на уровне парадигмы выделяется процедурный стиль, объектно ориентированный стиль (OO/классы) и функциональный стиль (FP).

- В процедурном стиле код представляет собой нисходящую линейную последовательность выполнения по заранее определенному набору операций, которые обычно объединяются в логически связанные единицы, называемые процедурами.
- В OO-стиле структура кода основана на группировке логики и данных в единицах, называемых классами.

- В функциональном стиле код структурируется по функциям (чистые вычисления в отличие от процедур), а адаптации этих функций становятся значениями.

Парадигмы не бывают правильными или ошибочными. Это ориентиры, которые направляют и формируют подход программиста к задачам и решениям, принципы структурирования и сопровождения кода. Некоторые языки ощутимо склоняются к одной парадигме — C относится к процедурным языкам, Java/C++ почти полностью ориентированы на работу с классами, а Haskell от начала до конца относится к функциональным.

Но многие языки также поддерживают паттерны, которые могут происходить (и даже смешиваться) из разных парадигм. Так называемые многопарадигменные языки обладают непревзойденной гибкостью. В некоторых случаях в одной программе даже могут соседствовать два и более разных выражения этих парадигм.

JavaScript, безусловно, относится к категории многопарадигменных языков. Вы можете писать процедурный, классово ориентированный или FP-код, причем эти решения могут приниматься на уровне отдельных строк — никто не заставляет вас действовать по принципу «все или ничего».

Прямая и обратная совместимость

Одним из самых фундаментальных принципов, по которым развивается JavaScript, является обеспечение *обратной совместимости*. Многих этот термин сбивает с толку, и они путают его с другим, хотя и похожим термином: *прямой совместимостью*.

Давайте расставим все по местам.

Обратная совместимость означает, что если нечто принимается как допустимый код JS, то в будущем не могут произойти изменения, из-за которых этот код станет недопустимым. Код, написанный в 1995 году, каким бы примитивным или ограниченным он ни был, должен работать и в наши дни. Как часто заявляют участники TC39, «мы не ломаем веб».

Суть в том, что разработчики JS могут писать код и быть полностью уверенными в том, что их код неожиданно не перестанет работать из-за обновления браузера. В результате решение о выборе JS для написания программы становится намного более разумным и надежным капиталовложением на многие будущие годы.

Не стоит недооценивать эту «гарантию». Сохранение обратной совместимости, растянувшееся почти на 25 лет истории языка, создает огромное бремя и обилие уникальных проблем. Вряд ли вы найдете в ком-

пьютерной отрасли много аналогичных примеров такого самоотверженного обеспечения обратной совместимости.

Также не стоит небрежно отмахиваться от затрат, связанных с соблюдением этого принципа. Он неизбежно устанавливает очень высокую планку для изменений и расширений языка; любое решение, вместе с ошибками и всем прочим, становится фактически необратимым. Когда что-то попало в JS, его уже нельзя убрать, потому что это может нарушить работоспособность программ, как бы сильно вам ни хотелось от этого избавиться!

У этого правила есть ряд небольших исключений. В JS было несколько изменений, не обладавших обратной совместимостью, но участники TC39 подходят к ним с исключительной осторожностью. Они анализируют существующий веб-код (посредством сбора информации о браузерах), чтобы оценить последствия такого нарушения, и в конечном счете именно разработчики браузеров решают и голосуют, хотят ли они столкнуться с недовольством пользователей из-за очень мелкой поломки ради выигрыша от исправления или усовершенствования некоторого аспекта языка для гораздо большего числа сайтов (и пользователей).

Такие изменения встречаются редко, и они почти всегда связаны с граничными ситуациями, которые

вряд ли приведут к заметному нарушению работы многих сайтов.

А теперь сравним обратную совместимость с ее двойником — прямой совместимостью. Прямая совместимость означает, что включение новой языковой возможности в программу не нарушит ее работоспособности, если она будет запущена в старом ядре JS. JS не обладает прямой совместимостью, хотя многие этого очень хотят и даже ошибочно верят в этот миф.

С другой стороны, HTML и CSS обладают прямой, но не обратной совместимостью. Если вы откопаете разметки HTML или CSS, написанные в 1995 году, вполне вероятно, что в наши дни они не будут работать (или будут работать иначе). Но если вы воспользуетесь новой возможностью от 2019 года в браузере, вышедшей в 2010 году, страница не «сломается» — нераспознанные фрагменты CSS/HTML пропускаются, тогда как остальные части обрабатываются как положено.

Может показаться, что прямую совместимость следовало бы включать в проектирование языка программирования, но обычно делать это нерационально. Разметка (HTML) и стилевое оформление (CSS) имеют декларативную природу, поэтому будет намного проще «пропустить» нераспознанные объявления с минимальными последствиями для других распознанных объявлений.

Но если язык программирования начнет избирательно пропускать непонятные команды (и даже выражения), это приведет к хаосу и недетерминированному поведению, потому что нельзя быть уверенным в том, что следующая часть программы не зависит от обработки пропущенной части.

Хотя JS не обладает (и не может обладать) прямой совместимостью, очень важно понимать суть обратной совместимости JS, включая долгосрочные преимущества для веб-страниц, а также ограничения и сложности, которые она создает для JS в результате.

Транспиляция

Так как JS не обладает прямой совместимостью, это означает, что всегда есть опасность разрыва между написанным вами допустимым кодом JS и самым старым ядром, которое должно поддерживаться вашим сайтом или приложением. Если вы запустите программу, использующую новые возможности ES2019 в ядре от 2016 года, скорее всего, это приведет к аварийному завершению программы.

Если речь идет о новом синтаксисе, то программа обычно просто отказывается компилироваться и запускаться — как правило, с выдачей синтаксической ошибки. Если же используется новый API (например, `Object.is(..)` из ES6), программа может отработать

до определенной точки, а потом выдать исключение времени выполнения и остановиться при обнаружении ссылки на неизвестный API.

Означает ли это, что разработчики JS обречены вечно отставать от технического прогресса, используя только устаревший код самого старого ядра JS, которое они должны поддерживать? Нет!

Но это означает, что разработчикам JS придется принимать специальные меры для преодоления этого разрыва.

Для нового и несовместимого синтаксиса проблема решается *транспиляцией*. Этот заумный термин, изобретенный в сообществе, описывает преобразование исходного кода программы из одной формы в другую (также в текстовый исходный код) специальной программой. Как правило, проблемы прямой совместимости, связанные с синтаксисом, решаются использованием транспилятора (самым распространенным из которых является Babel (https://babeljs.io)) для преобразования синтаксиса новой версии JS к эквивалентному старому синтаксису.

Например, разработчик может написать фрагмент кода следующего вида:

```
if (something) {
    let x = 3;
    console.log(x);
}
```

```
else {
    let x = 4;
    console.log(x);
}
```

Так этот код может выглядеть в дереве исходного кода приложения. Но при построении файла(-ов) для развертывания на общедоступном веб-сайте транспилятор Babel может преобразовать его к следующему виду:

```
var x$0, x$1;
if (something) {
    x$0 = 3;
    console.log(x$0);
}
else {
    x$1 = 4;
    console.log(x$1);
}
```

Исходный фрагмент зависел от использования `let` для создания переменных `x` в секциях `if` и `else`, которые не конфликтуют друг с другом. Эквивалентная программа (с минимальной переработкой), которую может сгенерировать Babel, просто присваивает переменным уникальные имена, обеспечивая тот же результат с исключением конфликта.

Возникает вопрос: зачем заморачиваться с преобразованием новой версии синтаксиса к старой при помощи специальной программы? Почему просто не использовать две переменные и обойтись без ключе-

вого слова `let`? Дело в том, что разработчикам настоятельно рекомендуется использовать новейшую версию JS, чтобы их код был чистым и наиболее эффективно передавал заложенные в нем идеи.

Ключевое слово `let` было добавлено в ES6 (в 2015 году). Предыдущий пример транспиляции актуален только в том случае, если приложение должно выполняться в среде JS до поддержки ES6. Здесь этот пример приведен только для простоты. Когда версия ES6 только появилась, транспиляция была весьма актуальной, но в 2020-м необходимость поддержки сред до ES6 встречается намного реже. Таким образом, «цель» транспиляции становится своего рода «скользящим окном», которое сдвигается вверх только при принятии решений о прекращении поддержки некоторого старого браузера/ядра сайтом или приложением.

Разработчики должны сосредоточиться на написании новых чистых синтаксических форм, а вся работа по построению версии кода с прямой совместимостью, пригодной для развертывания и запуска в средах самых старых поддерживаемых ядер JS, должна быть поручена программам.

Полифилы

Если проблема прямой совместимости связана не с новым синтаксисом, а с отсутствующим методом

API, который появился совсем недавно, обычно в таких ситуациях предоставляется определение для отсутствующего метода API, который работает так, словно он уже определен в более старой среде. Этот паттерн называется *полифилом* (polyfill, также shim).

Возьмем следующий код:

```
// getSomeRecords() возвращает промис для некоторого
// текста, который он получит
var pr = getSomeRecords();

// Вывод индикатора ожидания во время получения данных
startSpinner();

pr
.then(renderRecords)  // Вывести, если успешно.
.catch(showError)     // Вывести ошибку, если нет.
.finally(hideSpinner) // В любом случае скрыть
                      // индикатор ожидания.
```

В этом коде используется новая возможность ES2019 — метод `finally(..)` для прототипа промиса. Если вы попытаетесь использовать этот код в среде до ES2019, окажется, что метода `finally(..)` не существует, и произойдет ошибка.

Полифил для `finally(..)` в средах до ES2019 может выглядеть примерно так:

```
if (!Promise.prototype.finally) {
    Promise.prototype.finally = function f(fn){
        return this.then(
            function t(v){
```

```
                return Promise.resolve( fn() )
                    .then(function t(){
                        return v;
                    });
            },
            function c(e){
                return Promise.resolve( fn() )
                    .then(function t(){
                        throw e;
                    });
            }
        );
    };
}
```

Это всего лишь упрощенная иллюстрация простейшего (не полностью соответствующего спецификации) полифила для `finally(..)`. Не используйте его в своем коде; там, где это возможно, всегда используйте надежные официальные полифилы — например, коллекцию полифилов из ES-Shim.

Команда `if` защищает определение полифила, блокируя его выполнение в любой среде, в которой ядро JS уже определило этот метод. В более старых средах полифил определяется, но в новых средах команда `if` просто игнорируется.

Транспиляторы (например, Babel) обычно обнаруживают, какие полифилы нужны коду, и автоматически подставляют их за вас. Но время от времени приходится включать/определять их явно по аналогии с только что рассмотренным фрагментом.

Всегда пишите код с использованием средств, эффективно передающих его намерения и идеи. Как правило, это означает использование самой последней стабильной версии JS. Старайтесь не ухудшать удобочитаемость кода, пытаясь вручную заполнять пропуски в синтаксисе/API. Для этого есть программы!

Транспиляция и полифилы — два чрезвычайно эффективных приема для заполнения разрыва между кодом, использующим новейшие стабильные возможности языка, и старыми средами, которые должны поддерживаться сайтом или приложением. Так как JS не собирается останавливаться в своем развитии, этот разрыв будет присутствовать всегда. Оба приема должны стать стандартным звеном производственной цепочки каждого проекта JS.

Что такое интерпретация?

Давний вопрос относительно кода, написанного на JS: является ли он интерпретируемым сценарием или компилируемой программой? Похоже, большинство считает JS интерпретируемым языком (языком сценариев). Но истина несколько сложнее.

В истории языков программирования интерпретируемые языки и языки сценариев рассматривались как нечто менее полноценное по сравнению с их компилируемыми собратьями. Такое пренебрежи-

тельное отношение объясняется многими причинами, в том числе представлениями о недостаточной оптимизации на стадии выполнения и неприятия некоторых характеристик языка — например, языки сценариев обычно используют динамическую типизацию, в отличие от языков с «более зрелой» статической типизацией.

Языки, относимые к категории «компилируемых», обычно создают портируемое (двоичное) представление программы, которое распространяется для последующего выполнения. Так как обычно такое поведение для JS не характерно (распространяется исходный код, а не двоичная форма), многие утверждают, что это исключает JS из данной категории. На практике за последние десятилетия модель распространения программ в «исполняемой» форме стала намного более разнообразной, а также менее актуальной. Что касается обсуждаемого вопроса, в наше время уже не так важно, в какой именно форме распространяется программа.

Все эти необоснованные заявления и критику можно отложить в сторону. Настоящая причина, по которой важно четко понимать, является ли JS интерпретируемым или компилируемым языком, связана с природой обработки ошибок.

Исторически интерпретируемые языки (или языки сценариев) обычно выполнялись сверху вниз, строка

за строкой; как правило, исходный проход всей программы с обработкой кода до начала выполнения не применялся (рис. 1).

В языках сценариев или интерпретируемых языках ошибка в строке 5 программы не будет обнаружена до того момента, пока не будут выполнены строки с 1 по 4. В частности, ошибка в строке 5 может возникать из-за условия времени выполнения (например, некоторая переменная имеет недопустимое значение для операции) или присутствия некорректной команды в этой строке. В зависимости от контекста отложенная обработка ошибки до строки, в которой эта ошибка произошла, может быть как желательной, так и нежелательной.

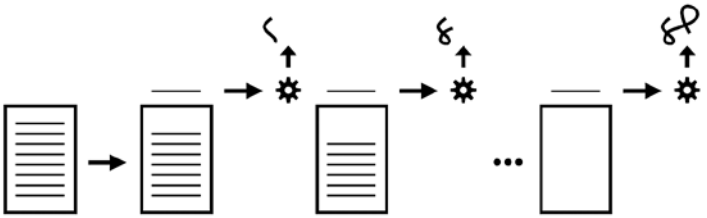


Рис. 1. Выполнение программ на интерпретируемом языке (языке сценариев)

Сравните с языками, которые проходят через этап предварительной обработки (разбора кода) до начала выполнения, как показано на рис. 2.

В этой модели обработки недействительная команда (например, некорректный синтаксис) в строке 5 будет перехвачен в фазе разбора, еще до выполнения какого-либо кода программы. Что касается синтаксических (или других статических) ошибок, о них желательно знать заранее, еще до неполного выполнения, обреченного на неудачу.

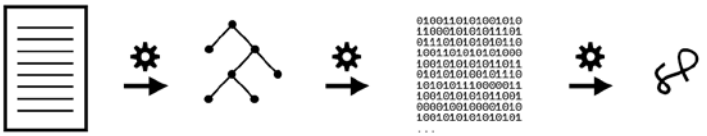


Рис. 2. Разбор + компиляция + выполнение

Что же общего у разбираемых языков с компилируемыми языками? Прежде всего, все компилируемые языки являются разбираемыми. Таким образом, разбираемый язык уже прошел немалую часть пути к статусу компилируемого. В классической теории компиляции последним шагом после разбора является генерирование кода, т. е. получение исполняемой формы.

После того как исходный код будет полностью разобран, последующее выполнение в том или ином виде обычно включает преобразование разобранной формы программы (часто называемой AST (Abstract Syntax Tree, т. е. «абстрактное синтаксическое дерево»)) в исполняемую форму.

Иначе говоря, разбираемые языки обычно также выполняют генерирование кода перед выполнением, поэтому не будет преувеличением сказать, что по духу они являются компилируемыми.

Исходный код JS разбирается перед выполнением. Этого требует спецификация, потому что в ней сказано, что «ранние ошибки» (статически определяемые ошибки в коде — такие, как совпадающие имена параметров) должны выявляться до начала выполнения кода. Такие ошибки не могут быть выявлены без разбора кода.

Итак, **JS является разбираемым языком**, но можно ли назвать его компилируемым?

Ответ ближе к «да», чем к «нет». Разобранный код JS преобразуется в оптимизированную (двоичную) форму, и именно этот код в дальнейшем выполняется (рис. 2); ядро обычно не переключается обратно в режим построчного выполнения (как на рис. 1) после всей тяжелой работы по разбору кода — по крайней мере этого не делает большинство языков/ядер, потому что это будет неэффективно.

Выражаясь конкретнее, эта компиляция производит двоичный байт-код (своего рода), который затем передается виртуальной машине JS для выполнения. Некоторые предпочитают говорить, что эта виртуальная машина интерпретирует байт-код. Но тогда это означает, что Java (и десятки других языков, находя-

щихся под управлением JVM) в этом отношении должны считаться интерпретируемыми, а не компилируемыми. Конечно, это противоречит типичному утверждению о том, что Java/и т. д. относятся к компилируемым языкам.

Интересно, что хотя языки Java и JavaScript очень сильно различаются, в отношении интерпретируемости/компилируемости между ними существует немалое сходство!

Другой нюанс заключается в том, что ядра JS могут применять к генерируемому коду многопроходную JIT-обработку/оптимизацию (Just-In-Time), которую также можно не без оснований назвать компиляцией или интерпретацией в зависимости от перспективы. На самом деле это невероятно сложная ситуация, скрываемая «под капотом» ядра JS.

К чему же сводятся все эти технические подробности? Сделаем шаг назад и рассмотрим всю последовательность выполнения исходного кода JS.

1. После того как программа выходит из редактора разработчика, она сначала транспилируется Babel, затем упаковывается Webpack (и, возможно, пятью-шестью другими процессами построения), после чего в совершенно новой форме передается ядру JS.

2. Ядро JS разбирает код в форму AST.

3. Ядро преобразует AST в своего рода байт-код — двоичное промежуточное представление (IR, Intermediate Representation), которое дополнительно совершенствуется/преобразуется оптимизирующим компилятором JIT.

4. Наконец, виртуальная машина JS выполняет программу.

Эти этапы наглядно представлены на рис. 3:

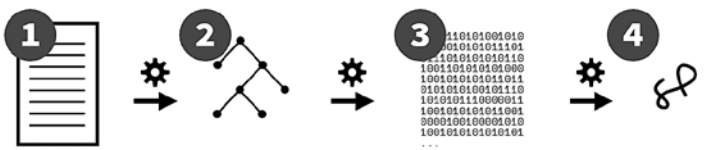


Рис. 3. Разбор, компиляция и выполнение JS

К чему ближе обработка JS — к интерпретируемому построчному сценарию, как на рис. 1, или к компилируемому языку, который перед выполнением обрабатывается в один или несколько проходов (как на рис. 2 и 3)?

На мой взгляд, очевидно, что по духу, если не в практическом смысле, **JS является компилируемым языком**.

Еще раз напомню, почему это важно: так как JS является компилируемым языком, мы получаем информацию о статических ошибках (например, некоррект-

ном синтаксисе) перед выполнением кода. Эта модель взаимодействий принципиально отличается от модели традиционных сценарных программ, и, пожалуй, она более эффективна.

WASM (Web Assembly)

Одним из факторов, значительно повлиявших на эволюцию JS, стало быстродействие — т. е. насколько быстро JS может разбираться/компилироваться и насколько быстро может выполняться откомпилированный код.

В 2013 году инженеры Mozilla Firefox продемонстрировали версию игрового движка Unreal 3, портированную с C на JS. Возможность выполнения этого кода в браузерном ядре JS при полных 60 кадрах в секунду основывалась на наборе оптимизаций, которые могли выполняться ядром JS именно потому, что в JS-версии кода Unreal использовался стиль программирования, который отдавал предпочтение специальному подмножеству языка JS под названием ASM.js.

Это подмножество содержит действительный код JS, написанный в стиле, несколько необычном для нормального программирования. Однако этот стиль передает ядру важную информацию типов, которая позволяет реализовать ключевые оптимизации ASM.js. Подмножество ASM.js создавалось как один из спо-

собов решения проблем быстродействия JS на стадии выполнения.

Важно заметить, что код ASM.js никогда не задумывался как код, написанный разработчиками. Это было представление программы, транспилированной с другого языка (например, C), в котором аннотации с информацией типов вставлялись автоматически инструментальной программой.

Через несколько лет после того, как код ASM.js продемонстрировал практичность генерируемых версий программ, которые могут более эффективно обрабатываться ядром JS, другая группа инженеров (которая изначально также происходила из Mozilla) опубликовала WASM (Web Assembly).

WASM напоминает ASM.js тем, что проект изначально был задуман для того, чтобы предоставить путь преобразования программ, не написанных на JS (например, на C и т. д.), в форму, которая может выполняться в ядре JS. В отличие от ASM.js, создатели WASM решили принять дополнительные меры для преодоления некоторых внутренних задержек, связанных с разбором/компиляцией JS, перед выполнением программы. Для этого программа представлялась в форме, которая не имела ничего общего с JS.

WASM — формат представления, отчасти напоминающий код ассемблера (отсюда название), который

может обрабатываться ядром JS. При этом пропускается фаза разбора/компиляции, обычно выполняемая ядром JS. Разбор/компиляция программ, предназначенных для преобразования в формат WASM, выполняется заранее (AOT, Ahead Of Time); при этом распространяется программа в двоичной форме, готовая для выполнения ядром JS с минимальной обработкой.

Очевидно, изначально формат WASM создавался для потенциального выигрыша по быстродействию. Хотя эта цель по-прежнему остается основной, дополнительную мотивацию WASM обеспечивало желание расширить возможности применения на веб-платформе других языков, кроме JS. Например, если язык Go поддерживает многопоточное программирование, а JS (язык) — нет, WASM обеспечивает возможности преобразования таких программ Go в форму, понятную для ядра JS, без необходимости поддержки потоков в самом языке JS.

Иначе говоря, WASM устраняет необходимость добавления в JS новых возможностей, в основном/исключительно предназначенных для использования транспилированными программами из других языков. Это означает, что развитие функциональности JS может оцениваться (комитетом TC39) без необходимости отвлекаться на интересы/потребности других языковых экосистем, сохраняя для других языков реальный путь на веб-платформу.

Интересно, что другая нарождающаяся точка зрения на WASM даже не имеет прямого отношения к веб-платформе (W). WASM эволюционирует и постепенно превращается в своего рода кроссплатформенную виртуальную машину (VM), на которой программы могут компилироваться однократно и выполняться в разных системных средах.

Таким образом, WASM существует не только для веб-платформ и не ограничивается JS.

Как ни парадоксально, хотя WASM работает в ядре JS, язык JS является одним из наименее подходящих для написания исходного кода программ WASM, потому что сильно зависит от статической информации о типах. Даже TypeScript (TS) — по сути JS + статические типы — не вполне подходят (на данный момент) для транспиляции в WASM, хотя разновидности языка (такие как AssemblyScript) пытаются заполнить пробел между JS/TS и WASM.

WASM не является главной темой этой книги, поэтому дальше обсуждать ее я не буду, но сделаю одно последнее замечание. Некоторые люди считают, что за WASM будущее, а JS исчезнет с веб-платформ или его использование будет сведено к минимуму. Такие люди часто испытывают неприязнь к JS и хотят, чтобы его заменил другой — любой другой! — язык. Так как WASM позволяет другим языкам работать в ядрах JS, на первый взгляд это не просто бесплодные мечты.

Но берусь утверждать: WASM не заменит JS. WASM значительно расширяет то, что может делать веб-платформа (включая JS). Это замечательно, но не имеет никакого отношения к тому, что некоторые люди используют WASM как путь к бегству от написания кода JS.

Строго говоря

В 2009 году при выходе ES5 в JS появился *строгий режим* (`strict`) как сознательно активизируемый механизм, способствующий повышению качества программ JS.

Преимущества строгого режима заметно перевешивают затраты, но старые привычки уходят не сразу, и инерцию существующих (унаследованных) кодовых баз достаточно трудно преодолеть. Грустно, но по прошествии более 10 лет *необязательность* строгого режима означает, что большинство программистов не считают его необходимым для себя по умолчанию.

Для чего нужен строгий режим? Не рассматривайте его как ограничение, мешающее что-то сделать. Скорее это проводник, который подсказывает наилучший способ выполнения тех или иных операций, чтобы ядро JS имело наибольшие шансы на оптимизацию и эффективное выполнение кода. Большая часть

кода JS разрабатывается в командах, так что обязательность строгого режима (вкупе с такими инструментами, как статические анализаторы) часто упрощает совместную работу над кодом, так как разработчики обходят некоторые опасные ошибки, которые могут встречаться в нестрогом режиме.

Большинство правил строгого режима имеет форму ранних ошибок, т. е. ошибок, которые формально не являются синтаксическими ошибками, но также выдаются во время компиляции (до выполнения кода). Например, строгий режим запрещает присваивать двум параметрам функции одинаковые имена; такая попытка приводит к ранней ошибке. Другие проявления строгого режима можно наблюдать только во время выполнения — например, как `this` по умолчанию имеет значение `undefined` вместо глобального объекта.

Вместо того чтобы спорить и протестовать против строгого режима (как ребенок, которому просто не нравятся родительские запреты), лучше всего относиться к строгому режиму как к статическому анализатору, который напоминает вам, как следует писать код JS для обеспечения наивысшего качества и оптимального быстродействия. Если вам кажется, что строгий режим ограничивает вашу свободу, и вы пытаетесь как-то обойти его, это должно быть красным сигналом тревоги: нужно сделать шаг назад и заново продумать подход к решению.

Строгий режим отключается на уровне файлов специальной директивой (перед которой не должно быть ничего, кроме комментариев/пропусков):

```
// до директивы use-strict
// разрешены только пропуски и комментарии
"use strict";
// остаток файла выполняется в строгом режиме
```

Учтите, что даже одиночный символ `;` перед директивой строгого режима делает ее бесполезной; никакие ошибки не выдаются, потому что строковый литерал в позиции команды является действительным кодом JS, однако при этом он приведет к незаметному отключению строгого режима!

Также есть возможность включения строгого режима на уровне отдельных функций, при этом действует то же правило относительно окружения:

```
function someOperations() {
    // здесь могут находиться пропуски и комментарии
    "use strict";
    // весь этот код будет выполняться в строгом режиме
}
```

Интересно, что если для файла включен строгий режим, то директивы строгого режима на уровне функций запрещаются. Поэтому нужно выбрать либо одно, либо другое.

Включение строгого режима на уровне функций может быть оправдано только в одном случае: если вы занимаетесь преобразованием существующей программы, в которой строгий режим отключен, и изменения вносятся небольшими порциями. В противном случае однозначно лучше просто включить строгий режим для всего файла/программы.

Многие интересовались, настанет ли время, когда в JS строгий режим будет использоваться по умолчанию? Отвечаю: почти безусловно нет. Как мы говорили ранее по поводу обратной совместимости, если обновленное ядро JS начнет считать, что код работает в строгом режиме, хотя он и не помечен как таковой, может оказаться, что работоспособность кода будет нарушена в результате действия строгого режима.

Однако некоторые факторы сокращают последствия того, что строгий режим не действует по умолчанию.

Во-первых, практически весь транспилированный код работает в строгом режиме, даже если исходный код не был написан в этом виде. Большая часть кода JS, находящегося в реальной эксплуатации, транспилируется; следовательно, большая часть кода JS уже работает в строгом режиме. Это предположение можно отменить, но для этого придется основательно потрудиться, так что это маловероятно.

Кроме того, есть распространенная тенденция к написанию большей части нового кода JS в формате модулей ES6. Модули ES6 подразумевают строгий режим, поэтому весь код в таких файлах использует строгий режим автоматически.

Подведем итог: в целом можно считать, что строгий режим является стандартом де-факто, хотя формально не используется по умолчанию.

После определения

JS является реализацией стандарта ECMAScript (версии 2019 на момент написания книги), который разрабатывается под руководством комитета TC39 и проводится по инициативе ECMA. Он работает в браузерах и других средах JS (например, Node.js).

JS является многопарадигменным языком; это означает, что его синтаксис и возможности позволяют разработчику смешивать концепции (а также сгибать и придавать им новую форму) из разных общепризнанных парадигм, включая процедурное, объектно-ориентированное и функциональное программирование.

JS является компилируемым языком в том смысле, что инструменты (включая ядро JS) обрабатывают и проверяют код программы (выдавая сообщения

о любых обнаруженных ошибках) перед ее выполнением.

Разобравшись с *определением* языка, начнем знакомство с его нюансами и хитростями.

2 Обзор возможностей JS

Лучший способ изучить JS — начать писать код JS.

Конечно, для этого нужно знать, как работает язык. На этой теме мы сосредоточимся в этой главе. Даже если вы уже программировали на других языках, постарайтесь без спешки освоиться с JS и обязательно потренируйтесь в использовании каждой составляющей.

Эта глава не является исчерпывающим справочником по всем аспектам синтаксиса языка JS. Также она не задумывалась как полноценный учебник «Введение в JS».

Вместо этого в ней приводится краткий обзор основных тематических разделов языка. Наша цель — поближе познакомиться с языком, чтобы вы могли перейти к написанию собственных программ с большей уверенностью. Многие из этих тем будут более подробно рассматриваться в оставшейся части этой и других книгах серии.

Если вы все еще продолжаете знакомиться с JS, рекомендую выделить побольше времени для работы над этой главой. Хорошо обдумайте каждый раздел и изучите тему. Просматривайте существующие программы JS; сравнивайте то, что вы увидите в них, с примерами и объяснениями (и мнениями), представленными в книге. При основательном понимании *природы* JS вы извлечете намного больше пользы из остального материала книги.

Не спешите, читая эту главу. Она длинная, и в ней приведено много подробностей, над которыми придется потрудиться. Будьте внимательны и не торопитесь.

Каждый файл является программой

Почти каждый сайт (веб-приложение), который вы используете, состоит из множества разных файлов JS (обычно имеющих расширение .js). Было бы соблазнительно представить себе их совокупность (приложение) как одну программу. Но с точки зрения JS это не так.

В JS каждый автономный файл представляет собой отдельную программу. Почему это важно? Главная причина связана с обработкой ошибок. Так как JS рассматривает файлы как программы, в одном файле может произойти ошибка (во время разбора/компиляции или выполнения), но это не обязательно помешает обработке следующего файла. Разумеется, если ваше приложение зависит от пяти файлов .js и в одном из них произойдет сбой, вероятно, приложение будет работать лишь частично (в лучшем случае). Важно проследить за тем, чтобы все файлы работали правильно и, насколько это возможно, корректно обрабатывали ошибки в других файлах.

Возможно, необходимость рассматривать разные файлы .js как отдельные программы вас удивит. С точки зрения использования приложения все выглядит как одна большая программа. Это связано с тем, что выполнение приложения позволяет этим отдельным программам взаимодействовать и работать как единое целое.

Многие программы используют средства построения, которые в конечном итоге объединяют разные файлы проекта в один файл, который должен предоставляться веб-странице. В таком случае JS рассматривает один объединенный файл как целую программу.

Несколько автономных файлов .js действуют как единая программа только в одном отношении: на уровне совместного доступа к их состоянию (и открытой функциональности) через глобальную область видимости. Они объединяются в пространстве имен глобальной области видимости и во время выполнения действуют как единое целое.

Начиная с ES6, JS также поддерживает формат модулей в дополнение к типичному формату автономных программ JS. Модули тоже базируются на файлах. Если файл загружается через механизм загрузки модулей, например командой `import` или тегом `<script type=module>`, то весь его код рассматривается как один модуль.

И хотя обычно мы не рассматриваем модуль — набор данных состояния и общедоступных методов для работы с ними — как автономную программу, JS в действительности обрабатывает каждый модуль по отдельности. По аналогии с тем, как глобальная область видимости позволяет объединять автономные файлы во время выполнения, импортирование модуля в другой модуль обеспечивает взаимодействие между ними во время выполнения.

Независимо от того, какой паттерн организации кода (и механизм загрузки) используется для файла (автономного или модуля), вы все равно должны представлять каждый файл как отдельную (мини-)программу, которая может взаимодействовать с другими (мини-)программами для выполнения функций общего приложения.

Значения

Наиболее фундаментальной единицей информации в программе является значение. Значения содержат данные. Они используются программами для хранения состояния. В JS значения существуют в двух формах: **примитивы** и **объекты**. Значения встраиваются в программы в виде *литералов*:

```
greeting("My name is Kyle.");
```

В этой программе значение `"My name is Kyle."` является примитивным строковым литералом; строки представляют собой упорядоченные наборы символов, которые обычно используются для представления слов и предложений.

Я воспользовался символом двойной кавычки " как ограничителем строкового значения. С таким же успехом можно было воспользоваться одинарной кавычкой '. Выбор зависит исключительно от личных предпочтений. Чтобы код лучше читался и не создавал проблем с сопровождением, важно выбрать один ограничитель и последовательно использовать его в программе.

Также в качестве ограничителя для строковых литералов может использоваться обратный апостроф `. Однако этот выбор уже не сводится к стилистическим предпочтениям; также изменяется поведение. Рассмотрим пример:

```
console.log("My name is ${ firstName }.");
// My name is ${ firstName }.

console.log('My name is ${ firstName }.');
// My name is ${ firstName }.

console.log(`My name is ${ firstName }.`);
// My name is Kyle.
```

Допустим, в этой программе уже определена переменная `firstName`, содержащая строковое значение

`"Kyle"`. В этом случае строка, заключенная в ограничители `` ` ``, заменяет переменное выражение (обозначенное символами `${ .. }`) его текущим значением. Такая замена называется *интерполяцией*.

Строка в ограничителях `` ` `` может использоваться без включения интерполируемых выражений, но это противоречит самой цели альтернативного синтаксиса строковых литералов:

```
console.log(
    `Am I confusing you by omitting interpolation?`
);
// Am I confusing you by omitting interpolation?
```

Правильное решение — использовать `"` или `'` (еще раз: выберите один вариант и придерживайтесь его!) для строк, если только вам не понадобится интерполяция; `` ` `` резервируется только для строк, которые будут включать интерполированные выражения.

Кроме строк, в программах JS часто встречаются другие примитивные литеральные значения — например, логические значения или числа:

```
while (false) {
    console.log(3.141592);
}
```

Ключевое `while` обозначает разновидность цикла — конструкции для повторения операций, пока условие остается истинным. В данном случае цикл не будет

выполнен ни разу (и ничего не будет выведено), потому что в условии цикла указано логическое значение `false`. Со значением `true` цикл будет выполняться бесконечно, так что будьте осторожны!

Число `3.141592`, как вам, вероятно, известно, является приближением математической константы π с точностью до шести знаков. Но вместо того чтобы встраивать это значение в программу, обычно рекомендуется использовать для этой цели заранее определенное значение `Math.PI`. Другой разновидностью числовых данных является примитивный тип `bigint`, предназначенный для хранения произвольных больших чисел.

Числовые данные часто используются в программах для отсчета — например, итераций цикла или обращения к информации в позиции, представленной числом (например, по индексу массива). Массивы/объекты будут рассмотрены позже, но если в программе существует массив с именем `names`, к элементу во второй позиции можно было бы обратиться следующим образом:

```
console.log(`My name is ${ names[1] }.`);
// My name is Kyle.
```

Для обозначения второй позиции использовано значение 1 вместо 2, потому что, как и в большинстве

языков программирования, нумерация индексов в массивах JS начинается с `0` (для первой позиции).

Кроме строк, чисел и логических значений в программах JS используются еще два *примитивных* значения: `null` и `undefined`. Хотя между ними существуют некоторые различия (как традиционные, так и современные), в основном обе служат одной цели — они обозначают пустое значение (т. е. отсутствие значения).

Многие разработчики предпочитают рассматривать их единообразно, т. е. эти значения считаются неразличимыми. Часто это оказывается возможно, если принять соответствующие меры. При этом надежнее и лучше использовать только `undefined` в качестве единственного пустого значения, хотя `null` и выглядит более привлекательно (приходится набирать меньше символов).

```
while (value != undefined) {
    console.log("Still got something!");
}
```

Последнее примитивное значение `Symbol` — специализированное значение, которое ведет себя как скрытое значение, которое невозможно угадать. Оно почти всегда используется исключительно в качестве специального ключа для объекта:

```
hitchhikersGuide[ Symbol("meaning of life") ];
// 42
```

В типичных программах JS примитивы `Symbol` используются не так часто. В основном они встречаются в низкоуровневом коде: библиотеках, фреймворках и т. д.

Массивы и объекты

Кроме примитивов в JS также используются объектные значения.

Как упоминалось ранее, массивы представляют собой особую разновидность объекта — упорядоченный список данных с числовыми индексами:

```
var names = [ "Frank", "Kyle", "Peter", "Susan" ];

names.length;
// 4

names[0];
// Frank

names[1];
// Kyle
```

Массивы JS позволяют хранить значения любых типов — как примитивы, так и объекты (включая другие массивы). Как будет показано в конце главы 3, даже функции являются значениями, которые могут храниться в массивах или объектах.

Функции, как и массивы, являются особой разновидностью (подтипом) объектов. Вскоре функции будут рассмотрены более подробно.

Объекты имеют более общую природу: они являются неупорядоченными наборами произвольных значений с доступом по ключу. Иначе говоря, вы обращаетесь к элементам по строковому имени (ключу или свойству) вместо числовой позиции (как в случае с массивами). Пример:

```
var me = {
    first: "Kyle",
    last: "Simpson",
    age: 39,
    specialties: [ "JS", "Table Tennis" ]
};

console.log(`My name is ${ me.first }.`);
```

Здесь `me` представляет объект, а `first` представляет имя, определяющее местонахождение информации в объекте (коллекции значений). Также для обращения информации в объекте может использоваться свойство/ключ в квадратных скобках:

```
me["first"]
```

Определение типа значения

Чтобы вы могли различать значения, оператор `typeof` возвращает встроенный тип значения для примитивов или `"object"` в противном случае:

```
typeof 42;                  // "number"
typeof "abc";               // "string"
typeof true;                // "boolean"
typeof undefined;           // "undefined"
typeof null;                // "object" -- ошибка!
typeof { "a": 1 };          // "object"
typeof [1,2,3];             // "object"
typeof function hello(){}; // "function"
```

К сожалению, `typeof null` возвращает `"object"` вместо ожидаемого `"null"`. Также `typeof` возвращает `"function"` для функций, но не возвращает `"array"` для массивов, как можно было бы предположить.

Преобразование между типами будет более подробно рассмотрено в этой главе.

Примитивные и объектные значения по-разному ведут себя при присваивании или передаче. Все эти нюансы рассматриваются в приложении А, раздел «Значения и ссылки».

Объявление и использование переменных

Пожалуй, стоит сказать то, что могло остаться неочевидным из предыдущего раздела: в программах JS значения либо присутствуют в виде литералов (как во многих предыдущих примерах), либо содержатся в переменных; переменные можно рассматривать как контейнеры для значений.

Чтобы переменная могла использоваться в программе, ее необходимо объявить (создать). Существуют различные синтаксические формы объявления переменных (идентификаторов), и каждая форма подразумевает свое поведение.

Для примера возьмем команду `var`:

```
var myName = "Kyle";
var age;
```

Ключевое слово `var` объявляет переменную для использования в этой части программы и допускает необязательное присваивание исходного значения.

Также существует похожее ключевое слово `let`:

```
let myName = "Kyle";
let age;
```

Ключевое слово `let` отличается от `var`. Самое очевидное различие заключается в том, что оно открывает более ограниченный доступ к переменной по сравнению с `var`. Это называется блоковой видимостью, в отличие от обычной (функциональной) видимости.

Пример:

```
var adult = true;

if (adult) {
    var myName = "Kyle";
    let age = 39;
    console.log("Shhh, this is a secret!");
}

console.log(myName);
// Kyle

console.log(age);
// Ошибка!
```

Попытка обратиться к `age` вне команды `if` приводит к ошибке, потому что переменная `age` имеет блоковую область видимости, которая ограничивается командой `if`, тогда как с переменной `myName` дело обстоит иначе.

Блоковая область видимости чрезвычайно удобна для ограничения распространения объявлений переменных в программах, так как она способствует предотвращению случайного перекрытия имен.

Однако ключевое слово `var` тоже полезно: оно означает «эта переменная должна быть видимой в более

широкой области видимости (всей функции)». Обе формы объявления могут оказаться уместными в той или иной части программы в зависимости от обстоятельств.

Очень часто приходится слышать, что от `var` следует полностью отказаться в пользу `let` (или `const`) — обычно из-за воображаемой путаницы с тем, как изменялось поведение `var` в отношении областей видимости с первых дней существования JS. Я считаю, что такой подход создает слишком серьезные ограничения и в конечном итоге бесполезен. Он предполагает, что вы не способны изучить и правильно использовать некоторую возможность языка в сочетании с другими возможностями. На мой взгляд, вы *можете* и *должны* изучать доступные возможности и использовать их там, где они уместны!

Третья форма объявления — `const`. Она похожа на `let`, но с дополнительным ограничением: ее значение должно быть задано в момент объявления и ей не может быть присвоено другое значение позднее.

Пример:

```
const myBirthday = true;
let age = 39;

if (myBirthday) {
    age = age + 1;      // OK!
    myBirthday = false; // Ошибка!
}
```

Изменение константы `myBirthday` и присваивание ей другого значения невозможно.

Переменные, объявленные ключевым словом `const`, не являются немодифицируемыми — просто им невозможно присвоить новое значение. Не рекомендуется использовать `const` с объектными значениями, потому что эти значения все равно могут изменяться, даже без повторного присваивания. В конечном итоге это приводит к потенциальной путанице, так что я считаю, что подобных ситуаций следует избегать:

```
const actors = [
"Morgan Freeman", "Jennifer Aniston"
];

actors[2] = "Tom Cruise"; // OK :(
actors = [];              // Ошибка!
```

Семантика использования `const` лучше всего подходит для ситуации, в которой у вас имеется одно примитивное значение, которому вы хотите присвоить полезное имя. Например, чтобы использовать `myBirthday` вместо `true`. Это упрощает чтение программ.

Если вы будете использовать `const` только с примитивными типами, вы сможете избежать любой путаницы с повторным присваиванием (запрещенным) и модификацией (разрешенной). Это самый надежный и лучший способ использования `const`.

Кроме var/let/const существуют и другие синтаксические формы, объявляющие идентификаторы (переменные) в различных областях видимости. Пример:

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

hello("Kyle");
// Hello, Kyle.
```

Идентификатор hello создается во внешней области видимости и при этом автоматически связывается со ссылкой на функцию. Но именованный параметр myName создается только внутри функции, и поэтому доступ к нему возможен только в области видимости функции. В целом hello и myName обычно ведут себя как объявленные ключевым словом var.

Также для объявления переменной может использоваться конструкция catch:

```
try {
    someError();
}
catch (err) {
    console.log(err);
}
```

Переменная err имеет блоковую область видимости, которая существует только внутри конструкции catch, как если бы она была объявлена ключевым словом let.

Функции

Слово «функция» имеет много смыслов в области программирования. Например, в мире функционального программирования термин «функция» имеет точное математическое определение и подразумевает жесткий набор правил, которые должны соблюдаться.

В JS смысл функции расширяется до другого взаимосвязанного термина: «процедура». Процедура представляет собой набор команд, который может вызываться один или несколько раз, может получать входные данные и может возвращать одно или несколько значений.

С первых дней существования JS определение функции выглядело так:

```
function awesomeFunction(coolThings) {
    // ..
    return amazingStuff;
}
```

Эта конструкция называется объявлением функции, потому что выглядит как самостоятельная команда, а не как выражение в другой команде.

Связь между идентификатором `awesomeFunction` и значением-функцией устанавливается в фазе компиляции кода, до его выполнения.

В отличие от команды объявления функции, определение и присваивание *функциональных выражений* могут выглядеть так:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

Эта функция является выражением, которое присваивается переменной `awesomeFunction`. В отличие от формы с объявлением функции, функциональное выражение не связывается с идентификатором до момента выполнения команды на стадии выполнения.

Очень важно понимать, что в JS функции являются значениями, которые могут присваиваться (как в этом фрагменте) и передаваться при вызове. Собственно функции JS составляют особую разновидность типов объектных значений. Не во всех языках функции рассматриваются как значения, но для языка очень важно поддерживать этот паттерн функционального программирования, как это делается в JS.

Функции JS могут получать входные данные в параметрах:

```
function greeting(myName) {
    console.log(`Hello, ${ myName }!`);
}

greeting("Kyle"); // Hello, Kyle!
```

В этом фрагменте идентификатор `myName` называется параметром; он действует как локальная переменная внутри функции. Функции могут определяться так, чтобы они получали любое количество параметров, от 0 и далее по вашему усмотрению. Каждому параметру присваивается значение-аргумент, которое передается в соответствующей позиции при вызове (`"Kyle"` в данном случае).

Функции также могут возвращать значения при помощи ключевого слова `return`:

```
function greeting(myName) {
    return `Hello, ${ myName }!`;
}

var msg = greeting("Kyle");

console.log(msg); // Hello, Kyle!
```

Вернуть можно только одно значение, но если вам потребовалось вернуть несколько значений, их можно упаковать в объект/массив. Так как функции являются значениями, их можно присваивать как свойства объекта:

```
var whatToSay = {
    greeting() {
        console.log("Hello!");
    },
    question() {
        console.log("What's your name?");
    },
```

```
    answer() {
        console.log("My name is Kyle.");
    }
};

whatToSay.greeting();
// Hello!
```

В этом фрагменте ссылки на три функции (`greeting()`, `question()` и `answer()`) включаются в объект, хранящийся под именем `whatToSay`. Каждую функцию можно вызвать, обратившись к свойству для получения значения — ссылки на функцию. Сравните этот прямолинейный стиль определения функций в объектах с более сложным синтаксисом классов, который будет рассматриваться позже в этой главе.

Функции принимают в JS много разных форм. Все эти вариации будут рассматриваться в приложении А, раздел «Многоликие функции».

Сравнения

Чтобы принимать решения в программах, необходимо сравнивать значения друг с другом, чтобы определить их отличительные признаки и отношения друг с другом. В JS предусмотрено несколько механизмов сравнения значений.

Равно... или типа того

Самое распространенное сравнение в программах JS отвечает на вопрос: «Значение X — *то же самое*, что значение Y?» Хотя что именно «то же самое» означает для JS?

По эргономическим и историческим причинам смысл сравнения сложнее, чем очевидное точное сопоставление тождественности. Иногда сравнение равенства подразумевает точное сопоставление, в других случаях за ним кроется более глубокий смысл, допускающий *достаточно близкое или взаимозаменяемое* сопоставление. Иначе говоря, необходимо помнить о нетривиальных различиях между сравнением на равенство и сравнением на эквивалентность.

Если вы уже работали с JS и читали литературу, наверняка вам встречался оператор `===`, также называемый оператором строгого равенства. Вроде бы смысл очевиден, не так ли? Наверняка под «строгим» имеется в виду «узкий и точный».

Не совсем так.

Да, большинство значений, задействованных в сравнении `===`, будут соответствовать интуитивным представлениям о точном совпадении. Несколько примеров:

```
3 === 3.0;                  // true
"yes" === "yes";            // true
```

```
null === null;              // true
false === false;            // true
42 === "42";                // false
"hello" === "Hello";        // false
true === 1;                 // false
0 === null;                 // false
"" === null;                // false
null === undefined;         // false
```

Проверка равенства оператором `===` часто описывается как проверка как значения, так и типа. В нескольких примерах, рассмотренных ранее (например, `42 === "42"`), типы двух значений (число, строка и т. д.) вроде бы становится определяющим фактором. Но этим дело не ограничивается. Тип сравниваемых значений учитывается во всех сравнениях значений в JS, не только в операторе `===`. А конкретнее, `===` запрещает любые преобразования типов при сравнении, тогда как при сравнениях JS они допустимы.

У оператора `===` есть некоторые нюансы, и многие разработчики JS забывают о них на свою беду. Оператор `===` сознательно проектировался так, чтобы возвращал ложную информацию для двух специальных значений: `NaN` и `-0`.

Пример:

```
NaN === NaN; // false
0 === -0;    // true
```

В случае `NaN` оператор `===` *врет*, что экземпляр `NaN` не равен другому `NaN`. В случае `-0` (да, это реальное значение, которое вы намеренно используете в своих программах) оператор `===` *врет*, что он равен обычному значению 0.

Так как *вранье* в таких сравнениях может создать проблемы, лучше не использовать `===` с этими значениями. Для сравнения `NaN` используется специальная функция `Number.isNaN(..)`, которая возвращает правдивый результат. Для сравнения `-0` лучше использовать функцию `Object.is(..)`, которая также говорит правду. При желании `Object.is(..)` также можно использовать для *правдивых* проверок `NaN`. Шутка: `Object.is(..)` можно рассматривать как оператор `====` (из четырех знаков равенства) — проверка настолько строгая, что строже некуда!

У этого вранья есть глубокие исторические и технические причины, но это не отменяет того факта, что `===` на самом деле не является *идеально строгой проверкой равенства* в *самом строгом* смысле.

История только усложняется, если мы рассмотрим сравнения объектных значений (непримитивов). Пример:

```
[ 1, 2, 3 ] === [ 1, 2, 3 ];  // false
{ a: 42 } === { a: 42 }       // false
(x => x * 2) === (x => x * 2) // false
```

Что тут происходит?

Казалось бы, разумно предположить, что проверка равенства учитывает *природу* или *содержимое* значения; в конце концов, `42 === 42` берет фактическое значение 42 и сравнивает его. Но когда речь заходит об объектах, сравнение с учетом содержимого обычно называется структурным равенством.

JS не определяет `===` как структурное равенство для объектных значений. Вместо этого `===` использует для объектных значений *тождественное равенство*.

В JS все объектные значения хранятся по ссылке (см. приложение A, «Значения и ссылки»), присваиваются и передаются копированием ссылки, а в контексте нашего текущего обсуждения сравниваются по ссылочному (тождественному) равенству. Пример:

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

// Присваивание выполняется копированием ссылки,
// поэтому y ссылается на *тот же* массив, что и x,
// а не на его новую копию.
var y = x;

y === x; // true
y === [ 1, 2, 3 ]; // false
x === [ 1, 2, 3 ]; // false
```

В этом фрагменте выражение `y === x` истинно, потому что обе переменные содержат ссылку на один и тот же исходный массив. Но оба сравнения `===`

`[1,2,3]` завершаются неудачей, потому что `y` и `x` соответственно сравниваются с новыми *разными* массивами `[1,2,3]`. Структура массива и содержимое в данном случае роли не играют — важна только **тождественность ссылок**.

JS не предоставляет механизм проверки структурного равенства объектных значений — только проверку тождественности ссылок. Чтобы выполнить проверку структурного равенства, придется реализовать проверки самостоятельно.

Но учтите, что эта задача сложнее, чем может показаться. Например, как определить, что две ссылки на функции структурно эквивалентны? Даже если преобразовать их в строковую форму для сравнения исходного кода, при этом не будут учитываться такие аспекты, как замыкания. JS не обеспечивает сравнения структурного равенства, потому что учесть все граничные случаи практически невозможно!

Сравнения с преобразованием типа

Преобразование типа означает, что значение одного типа преобразуется в относительное представление другого типа (насколько это возможно). Как будет показано в главе 4, преобразование типа является одним из столпов языка JS, а не каким-то дополнительным инструментом, которого можно избежать.

К сожалению, когда преобразование типов сталкивается с операторами сравнения (например, проверки равенства), путаница и разочарования возникают гораздо чаще, чем хотелось бы.

Мало какие возможности JS вызывают больше раздражения в сообществе JS, чем оператор ==, который обычно называют оператором свободного равенства. Как правило, этот оператор осуждается как плохо спроектированный и опасный/ненадежный в программах JS. Даже сам создатель языка Брендан Эйх жаловался, что он был спроектирован крайне неудачно.

Насколько я могу судить, большая часть всех этих недоразумений происходит из-за довольно короткого списка запутанных граничных случаев, но существует и более глубокая проблема: чрезвычайно распространенное заблуждение о том, что оператор выполняет сравнения без учета типа сравниваемых значений.

Оператор == выполняет проверку равенства аналогично тому, как ее выполняет оператор ===. В сущности, оба оператора учитывают тип сравниваемых значений. И если сравниваются значения одинаковых типов, == и === работают абсолютно одинаково, вообще без каких-либо различий.

Если же сравниваются разные типы, то == отличается от === тем, что он допускает преобразование типа

перед сравнением. Иначе говоря, оба стараются сравнивать значения похожих типов, но `==` позволяет *сначала* выполнить преобразование типа, и после того как в результате преобразования с обеих сторон стоят значения одинаковых типов, `==` делает то же самое, что и `===`. Вместо свободного равенства оператор `==` правильнее было бы называть равенством с преобразованием типа.

Пример:

```
42 == "42"; // true
1 == true;  // true
```

В обоих сравнениях типы значений различны, так что оператор `==` преобразует нечисловые значения (`"42"` и `true`) в числа перед сравнением.

Даже если вы просто будете знать об этой особенности `==` (что оператор отдает предпочтение примитивным числовым значениям), это поможет избежать большинства неприятных граничных случаев — например, вы будете держаться подальше от таких потенциальных ловушек, как `"" == 0` или `0 == false`.

Возможно, вы подумали: «Ну и ладно, я просто навсегда откажусь от проверки равенства с преобразованием типа (и буду использовать `===`), чтобы избежать этих граничных случаев!» К сожалению, у вас это вряд ли получится.

Скорее всего, вы будете использовать в программах операторы относительного сравнения, такие как `<`, `>` (и даже `<=` и `>=`).

Как и `==`, эти операторы ведут себя строго, если сравниваемые типы уже совпадают, но позволяют сначала выполнить преобразование (обычно в число) при различающихся типах.

Пример:

```
var arr = [ "1", "10", "100", "1000" ];
for (let i = 0; i < arr.length && arr[i] < 500; i++) {
   // will run 3 times
}
```

Сравнение `i < arr.length` защищено от преобразования типа, потому что `i` и `arr.length` всегда являются числами. Однако сравнение `arr[i] < 500` активизирует преобразование типа, потому что все значения `arr[i]` являются строками. Таким образом, сравнения превращаются в `1 < 500`, `10 < 500`, `100 < 500` и `1000 < 500`. Так как четвертое условие ложно, цикл останавливается после третьей итерации.

Операторы относительного сравнения обычно используют сравнения чисел, кроме того случая, в котором оба сравниваемых значения уже являются строками; в таком случае используется алфавитное сравнение строк:

```
var x = "10";
var y = "9";

x < y; // осторожно, true!
```

Заставить операторы относительного сравнения обходиться без преобразования типов можно только одним способом: никогда не использовать несовпадающие типы в сравнениях. Возможно, эта цель достойна восхищения, и все же с довольно большой вероятностью вы столкнетесь со случаем, в котором типы могут различаться.

Разумнее не избегать сравнений с преобразованием типов, но принять их и изучить все тонкости их использования.

Сравнения с преобразованием типов встречаются и в других местах JS — например, в условных командах (`if` и т. д.), к которым мы еще вернемся в приложении А, раздел «Условные сравнения с преобразованием типов».

Организация кода JS

В экосистеме JS широко применяются два основных паттерна организации кода (данных и поведения): классы и модули. Эти паттерны не являются взаимоисключающими: во многих программах можно и нужно использовать оба. Другие программы могут при-

держиваться одним паттерном или даже обойтись вообще без них!

В некоторых отношениях эти паттерны сильно различаются. Но, как ни странно, в чем-то они всего лишь являются разными сторонами одной медали. Чтобы профессионально владеть JS, вы должны понимать оба паттерна и знать, где их уместно использовать (а где нет).

Классы

Термины «объектно-ориентированный», «классово-ориентированный» и «классы» чрезвычайно сильно перегружены подробностями и нюансами; у них нет единственно верного определения.

Мы будем использовать распространенное и отчасти традиционное определение. Скорее всего, оно наиболее знакомо читателям с опытом работы на объектно-ориентированных языках вроде C++ и Java.

Класс в программе является определением типа специальной структуры данных, включающей как данные, так и поведение, работающее с этими данными. Классы определяют, как работает такая структура данных, но сами они не являются конкретными значениями. Чтобы получить конкретное значение, которое можно использовать в программе, необходимо создать экземпляр класса (при помощи ключевого слова `new`) один или несколько раз.

Пример:

```
class Page {
    constructor(text) {
        this.text = text;
    }
    print() {
        console.log(this.text);
    }
}

class Notebook {
    constructor() {
        this.pages = [];
    }

    addPage(text) {
        var page = new Page(text);
        this.pages.push(page);
    }

    print() {
        for (let page of this.pages) {
            page.print();
        }
    }
}

var mathNote = new Notebook();
mathNotes.addPage("Arithmetic: + - * / ...");
mathNotes.addPage("Trigonometry: sin cos tan ...");

mathNotes.print();
// ..
```

В классе `Page` данные представляют собой строку текста, хранящуюся в свойстве `this.text`. Поведение

составляет `print()` — метод для вывода текста на консоль.

Для класса `Notebook` данные представлены массивом экземпляров `Page`. К поведению относится метод `addPage(..)`, который создает экземпляры `Page` и добавляет их в список, а также метод `print()`, который выводит все содержимое экземпляров `Page` из списка.

Команда `mathNotes = new Notebook()` создает экземпляр класса `Notebook`, а в команде `page = new Page(text)` создаются экземпляры класса `Page`. Методы (поведение) могут вызываться только для экземпляров (но не для самих классов) — например, `mathNotes.addPage(..)` и `page.print()`.

Механизм `class` позволяет упаковать данные (`text` и `pages`) вместе с поведением (например, `addPage(..)` и `print()`). Ту же программу можно было построить без определений классов, но, скорее всего, она получилась бы намного менее организованной, хуже читалась, была бы менее понятной и более подверженной ошибкам и на ее сопровождение потребовалось бы слишком много сил.

Наследование классов

Другой аспект, присущий традиционному классово-ориентированному проектированию, хотя и реже применяемый в JS, — наследование (и полиморфизм). Пример:

```
class Publication {
    constructor(title,author,pubDate) {
        this.title = title;
        this.author = author;
        this.pubDate = pubDate;
    }

    print() {
        console.log(`
            Title: ${ this.title }
            By: ${ this.author }
            ${ this.pubDate }
        `);
    }
}
```

Класс `Publication` определяет набор общих аспектов поведения, которые могут понадобиться для любой публикации.

А теперь рассмотрим более конкретные типы публикаций, такие как `Book` и `BlogPost`:

```
class Book extends Publication {
    constructor(bookDetails) {
        super(
            bookDetails.title,
            bookDetails.author,
            bookDetails.publishedOn
        );
        this.publisher = bookDetails.publisher;
        this.ISBN = bookDetails.ISBN;
    }

    print() {
        super.print();
```

```
        console.log(`
             Publisher: ${ this.publisher }
             ISBN: ${ this.ISBN }
        `);
    }
}

class BlogPost extends Publication {
    constructor(title,author,pubDate,URL) {
        super(title,author,pubDate);
        this.URL = URL;
    }

    print() {
        super.print();
        console.log(this.URL);
    }
}
```

В объявлении `Book` и `BlogPost` используется ключевое слово `extends`. Оно означает, что классы расширяют общее определение `Publication` для включения в него дополнительного поведения. Вызов `super(..)` в каждом конструкторе активизирует конструктор родительского класса `Publication` для выполнения его части инициализации, после чего выполняет специфические операции для соответствующего вида публикации (обозначаемого термином «подкласс», или «производный класс»).

Пример использования подклассов:

```
var YDKJS = new Book({
    title: "You Don't Know JS",
```

```
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'Reilly",
    ISBN: "123456-789"
});

YDKJS.print();
    // Title: You Don't Know JS
    // By: Kyle Simpson
    // June 2014
    // Publisher: O'Reilly
    // ISBN: 123456-789

var forAgainstLet = new BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
    // Title: For and against let
    // By: Kyle Simpson
    // October 27, 2014
    // https://davidwalsh.name/for-and-against-let
```

Обратите внимание: оба экземпляра подкласса содержат метод `print()`, который был переопределением метода `print()`, *унаследованного* от родительского класса `Publication`. Каждый из переопределенных методов `print()` в подклассах вызывает `super.print()` для передачи управления унаследованной версии метода `print()`.

Тот факт, что унаследованный и переопределенный методы могут иметь одинаковые имена и сосуществовать в классе, называется *полиморфизмом*.

Наследование — мощный инструмент для организации данных/поведения в отдельных логических единицах (классах) так, что подкласс может взаимодействовать с родительским классом, обращаться к его поведению/данным и использовать их.

Модули

Паттерн модуля фактически преследует ту же цель, что и паттерн класса: он предназначен для группировки данных и поведения в логических единицах. Кроме того, как и в классах, модули могут «включать» или «обращаться» к данным и поведению других модулей с целью взаимодействия.

Тем не менее модули отличаются от классов в нескольких важных отношениях. Самое заметное отличие — совершенно иной синтаксис.

Классические модули

В ES6 к исходному синтаксису JS (который вскоре будет рассмотрен) была добавлена синтаксическая форма модулей. Однако с первых дней существования JS модули были важным и распространенным паттер-

ном, который был задействован в бесчисленных программах JS даже без специального синтаксиса.

Ключевые признаки *классического модуля* — внешняя функция (выполняемая как минимум один раз), которая возвращает экземпляр модуля с одной или несколькими функциями, способными работать с внутренними (скрытыми) данными экземпляра модуля.

Так как модуль в этой форме — *всего лишь функция*, вызов которой создает экземпляр модуля, такие функции также описываются как фабрики модулей.

Рассмотрим классическую форму модуля из приведенных ранее классов `Publication`, `Book` и `BlogPost`:

```
function Publication(title,author,pubDate) {
    var publicAPI = {
        print() {
            console.log(`
                Title: ${ title }
                By: ${ author }
                ${ pubDate }
            `);
        }
    };

    return publicAPI;
}

function Book(bookDetails) {
    var pub = Publication(
        bookDetails.title,
        bookDetails.author,
        bookDetails.publishedOn
```

```
    );

    var publicAPI = {
        print() {
            pub.print();
            console.log(`
                Publisher: ${ bookDetails.publisher }
                ISBN: ${ bookDetails.ISBN }
            `);
        }
    };

    return publicAPI;
}

function BlogPost(title,author,pubDate,URL) {
    var pub = Publication(title,author,pubDate);

    var publicAPI = {
        print() {
            pub.print();
            console.log(URL);
        }
    };

    return publicAPI;
}
```

Сравните эти формы с формами `class`; сходства между ними едва ли не больше, чем различий.

Форма `class` хранит методы и данные в экземпляре объекта, для обращения к которому используется префикс `this.`. С модулями к методам и данным можно обращаться как к переменным в области видимости без префикса `this.`.

С формой `class` API экземпляра неявно присутствует в определении класса — кроме того, все данные и методы являются открытыми. С функцией фабрики модуля вы явно создаете и возвращаете объект с открытыми методами, а все данные и другие не включенные методы остаются приватными внутри фабричной функции.

У формы фабричной функции существуют и другие разновидности, достаточно распространенные в JS даже в 2020 году. Их можно встретить в самых разных программах JS: AMD (Asynchronous Module Definition), UMD (UniversalModule Definition), CommonJS (классические модули в стиле Node.js). Эти разновидности не полностью совместимы. Но все эти формы базируются на одних принципах.

Рассмотрим использование (т. е. создание экземпляра) этих фабричных функций модулей:

```
var YDKJS = Book({
    title: "You Don't Know JS",
    author: "Kyle Simpson",
    publishedOn: "June 2014",
    publisher: "O'Reilly",
    ISBN: "123456-789"
});

YDKJS.print();
// Title: You Don't Know JS
// By: Kyle Simpson
// June 2014
// Publisher: O'Reilly
```

```
// ISBN: 123456-789

var forAgainstLet = BlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

Единственное заметное отличие — отсутствие `new` и вызов фабрик модулей как обычных функций.

Модули ES

Модули ES (ESM), включенные в язык JS в ES6, должны служить практически тем же целям, что и только что описанные *классические модули*, особенно с учетом различий в важных нюансах и сценариях использования из AMD, UMD и CommonJS.

Тем не менее подход к реализации сильно отличается.

Во-первых, не существует функции-обертки для *определения* модуля. Контекстом обертки является файл. Модули ESM всегда базируются на файлах: один файл — один модуль.

Во-вторых, не обязательно явно взаимодействовать с API модуля; достаточно воспользоваться ключевым словом `export`, чтобы добавить переменную или метод в его определение открытого API. Если нечто определяется в модуле, но не экспортируется, то оно остается скрытым (как и с *классическими модулями*).

В-третьих, вы не создаете экземпляр модуля ES, а просто импортируете его для использования его единственного экземпляра (пожалуй, это самое заметное отличие от паттернов, упоминавшихся ранее). По сути, модули ESM являются одиночками[1] — в программе они существуют только в единственном экземпляре, который создается при первом импортировании, а все последующие команды импортирования просто получают ссылку на тот же экземпляр. Если ваш модуль должен существовать в нескольких экземплярах, то придется предоставить фабричную функцию в стиле классических модулей для вашего определения ESM.

Наш текущий пример предполагает создание нескольких экземпляров, поэтому в следующих фрагментах будут параллельно использоваться как модули ESM, так и классические модули.

[1] Одиночка (Singleton) — порождающий паттерн проектирования. — *Примеч. ред.*

Возьмем файл publication.js:

```
function printDetails(title,author,pubDate) {
    console.log(`
        Title: ${ title }
        By: ${ author }
        ${ pubDate }
    `);
}

export function create(title,author,pubDate) {
    var publicAPI = {
            print() {
            printDetails(title,author,pubDate);
        }
    };

    return publicAPI;
}
```

Импортирование и использование этого модуля из другого модуля ES — например, blogpost.js:

```
import { create as createPub } from "publication.js";

function printDetails(pub,URL) {
    pub.print();
    console.log(URL);
}

export function create(title,author,pubDate,URL) {
    var pub = createPub(title,author,pubDate);

    var publicAPI = {
        print() {
```

```
            printDetails(pub,URL);
        }
    };
    return publicAPI;
}
```

И наконец, чтобы использовать этот модуль, мы импортируем его в другой модуль ES — например, main.js:

```
import { create as newBlogPost } from "blogpost.js";

var forAgainstLet = newBlogPost(
    "For and against let",
    "Kyle Simpson",
    "October 27, 2014",
    "https://davidwalsh.name/for-and-against-let"
);

forAgainstLet.print();
// Title: For and against let
// By: Kyle Simpson
// October 27, 2014
// https://davidwalsh.name/for-and-against-let
```

Секция `as newBlogPost` в команде `import` необязательна; если опустить ее, то будет импортирована только функция верхнего уровня с именем `create(..)`. В данном случае я переименовал ее для удобства чтения; более общее имя фабрики `create(..)` заменяется более семантически содержательным `as newBlogPost(..)`.

Как показано, модули ES могут использовать *классические модули* в своей внутренней работе, если им необходимо поддерживать создание множественных экземпляров. Также можно было бы предоставить доступ к классу из модуля вместо фабричной функции `create(..)`, обычно с тем же результатом. Но поскольку к этому моменту вы уже используете ESM, я бы порекомендовал придерживаться *классических модулей* вместо класса.

Если вашему модулю необходим только один экземпляр, дополнительные уровни сложности можно обойти: просто экспортируйте его открытые методы напрямую.

Кроличья нора становится глубже

Как я обещал в начале этой главы, мы всего лишь бросили беглый взгляд на поверхность основных частей языка JS. Возможно, у вас голова идет кругом, но это абсолютно нормально после такого обилия информации.

Даже в этом кратком обзоре JS мы описали или указали на многочисленные подробности, которые необходимо тщательно учитывать и хорошенько изучить. Настоятельно советую перечитать эту главу. Можно даже несколько раз.

В следующей главе мы намного глубже рассмотрим некоторые фундаментальные аспекты работы JS. Но прежде чем спускаться дальше в кроличью нору, не жалейте времени и убедитесь в том, что вы в полной мере усвоили весь материал этой главы.

3 JS: копаем вглубь

Если вы прочитали главы 1 и 2 и не пожалели времени на то, чтобы хорошенько обдумать материал, скорее всего, вы уже начали понимать JS чуть лучше. Если же вы пропустили или бегло пролистали их (особенно главу 2), то рекомендую вернуться и еще поработать с этим материалом.

В главе 2 синтаксис, паттерны и поведение рассматривались на высоком уровне. В этой главе мы переключимся на низкоуровневые характеристики JS, которые пролегают практически под каждой написанной вами строкой кода.

Учтите: в этой главе вы столкнетесь с гораздо более глубокими концепциями JS, о которых вы, возможно, не задумывались прежде. Я постараюсь помочь вам разобраться в том, как JS работает на фундаментальном уровне, что приводит его в движение. Здесь вы получите ответ на некоторые «почему?», которые, возможно, возникали у вас в процессе исследования JS. Впрочем, этот материал все равно не дает исчерпывающего описания языка; для этого понадобятся остальные книги этой серии!

Наша цель в этой главе — *приступить к изучению* и проникнуться *атмосферой* JS, его сильными и слабыми сторонами.

Не пытайтесь наскоком взять этот материал, иначе вы рискуете заблудиться в зарослях. Как я уже неоднократно говорил, не жалейте времени. Впрочем,

даже после чтения этой главы у вас, скорее всего, останутся вопросы. И это нормально, ведь вас ждет еще целая серия книг, с которой вы сможете продолжить свои исследования!

Итерации

Так как программы по сути строятся для обработки данных (и принятия решений на основании этих данных), паттерны, применяемые для пошагового перебора данных, сильно влияют на удобочитаемость программы.

Паттерн «Итератор» существует уже несколько десятилетий и предполагает стандартизированный подход к потреблению данных из источника, порцию за порцией. Идея заключается в том, что удобнее и практичнее обрабатывать источник — т. е. коллекцию данных — последовательно: сначала первую часть, потом вторую и т. д., вместо того чтобы работать со всем набором сразу.

Вообразите структуру данных, которая представляет собой запрос `SELECT` к реляционной БД. В таких структурах результаты обычно выдаются в виде строк данных. Если запрос возвращает одну-две строки данных, можно работать со всем итоговым набором сразу, присвоить каждую строку локальной перемен-

ной и выполнить с данными любые требуемые операции.

Но если запрос состоит из 100 или 1000 (а то и более!) строк, для работы с данными придется использовать итеративную обработку (как правило, цикл).

Паттерн определяет структуру данных, которая называется итератор и содержит ссылку на нижележащий источник данных (например, строки данных результата запроса), который предоставляет метод `next()` (или другой метод с похожим именем). Вызов `next()` возвращает следующий фрагмент данных (запись или строку данных из запроса).

Не всегда известно заранее, сколько фрагментов данных вам придется перебрать в ходе итераций, поэтому в паттерне завершение перебора обычно обозначается каким-нибудь специальным значением или исключением, которое выдается после завершения всего набора и *выхода за его границу*.

Важность паттерна «Итератор» заключается в стандартном способе итеративной обработки данных, который создает более понятный и доступный код, вместо того чтобы каждая структура данных/источник определяла собственный способ обработки своих данных.

После многолетних усилий сообщества JS, направленных на создание общепринятых методов итерации, в ES6 был стандартизирован конкретный протокол

для паттерна «Итератор» прямо в языке. Протокол определяет метод `next()`, который возвращает объект, называемый *результатом итератора*; объект имеет свойства `value` и `done`, где `done` — логическое значение, равное `false` до того, как перебор по нижележащему источнику данных не будет завершен.

Потребление итераторов

При наличии протокола итераторов ES6 появилась возможность потребления источников данных по одному значению; после каждого вызова `next()` свойство `done` проверяется на истинность для прекращения итераций. Но в таком решении слишком многое приходится делать вручную, поэтому ES6 также включает несколько механизмов (на уровне синтаксиса и API) для стандартизированного потребления этих итераторов.

Одним из таких механизмов является цикл `for..of`:

```
// Имеется итератор для некоторого источника данных:
var it = /* .. */;

// последовательный перебор его результатов
for (let val of it) {
    console.log(`Iterator value: ${ val }`);
}

// Iterator value: ..
// Iterator value: ..
// ..
```

Эквивалентный цикл с «ручной» реализацией я приводить не буду, но он безусловно читается хуже, чем цикл `for..of`.

Другой механизм, часто используемый для потребления итераторов, — оператор `...`. Этот оператор существует в двух симметричных формах: распределения (spread) и остатка (rest). Форма распределения используется для потребления итераторов.

Чтобы распределить итератор, необходимо иметь структуру для размещения распределенных компонентов. В JS предусмотрены две возможности: массив и список аргументов для вызова функции.

Распределение в массив:

```
// распределение итератора в массив,
// каждое значение в переборе занимает
// отдельный элемент (позицию) в массиве.
var vals = [ ...it ];
```

Распределение при вызове функции:

```
// распределение итератора в вызов функции,
// каждое значение в переборе занимает
// отдельный аргумент (позицию).
doSomethingUseful( ...it );
```

В обоих случаях форма распределения `...` следует протоколу потребления итератора (такому же, как для цикла `for..of`) для получения всех доступных

значений от итератора и размещения (распределения) их в получающем контексте (массив, список аргументов).

Итерируемые значения

С технической точки зрения протокол потребления итератора определяется для потребления *итерируемых значений* (iterables); итерируемое значение представляет собой значение, перебор содержимого которого может осуществляться при помощи итератора.

Протокол автоматически создает экземпляр итератора по итерируемому значению и потребляет только этот экземпляр итератора до его завершения. Это означает, что одно итерируемое значение может потребляться многократно; при этом каждый раз создается и используется новый экземпляр итератора.

Где же взять итерируемые значения?

В ES6 были определены базовые типы структур данных/коллекций JS, которые могут использоваться как итерируемые значения. К их числу относятся строки, массивы, карты, множества и т. д.

Пример:

```
// массив является итерируемым значением
var arr = [ 10, 20, 30 ];
```

```
for (let val of arr) {
    console.log(`Array value: ${ val }`);
}
// Array value: 10
// Array value: 20
// Array value: 30
```

Так как массивы являются итерируемыми значениями, для поверхностного копирования массива может использоваться потребление итератора с оператором `...`:

```
var arrCopy = [ ...arr ];
```

Также оператор может использоваться для последовательного перебора символов строки:

```
var greeting = "Hello world!";
var chars = [ ...greeting ];

chars;
// [ "H", "e", "l", "l", "o", " ",
// "w", "o", "r", "l", "d", "!" ]
```

Структура данных `Map` (карта, словарь, ассоциативный массив) использует объекты в качестве ключей; с этими объектами связываются значения (произвольного типа). По умолчанию `Map` использует не такую итерацию, как показано здесь, — перебираются не значения карты, а *записи*. *Запись* представляет собой кортеж (массив из 2 элементов), включающий как ключ, так и значение.

Пример:

```
// Два элемента DOM, `btn1` и `btn2`
var buttonNames = new Map();
buttonNames.set(btn1,"Button 1");
buttonNames.set(btn2,"Button 2");

for (let [btn,btnName] of buttonNames) {
    btn.addEventListener("click",function onClick(){
        console.log(`Clicked ${ btnName }`);
    });
}
```

В цикле `for..of`, использующем перебор карты по умолчанию, синтаксис `[btn,btnName]` (так называемая деструктуризация массива) используется для разбиения всех потребляемых кортежей на пары «ключ — значение» (`btn1` / `"Button 1"` и `btn2` / `"Button 2"`).

Каждое из встроенных итерируемых значений в JS предоставляет механизм перебора по умолчанию, который, скорее всего, будет соответствовать вашим интуитивным представлениям. Но при необходимости также можно выбрать более конкретный механизм перебора. Например, если вы хотите потреблять только значения из приведенной выше карты `buttonNames`, можно вызвать `values()` для получения итератора, перебирающего только значения:

```
for (let btnName of buttonNames.values()) {
    console.log(btnName);
}
// Button 1
// Button 2
```

А если вы хотите получить индекс *и* значение для перебора массива, создайте итератор для записей методом `entries()`:

```
var arr = [ 10, 20, 30 ];

for (let [idx,val] of arr.entries()) {
    console.log(`[${ idx }]: ${ val }`);
}
// [0]: 10
// [1]: 20
// [2]: 30
```

По большей части для всех встроенных итерируемых значений в JS доступны три формы итераторов: только для ключей (`keys()`), только для значений (`values()`) и для записей (`entries()`).

Возможно, вы обратили внимание на небольшую смену курса в этом обсуждении. Мы начали с потребления итераторов, но затем переключились на обсуждение перебора по итерируемым значениям. Протокол потребления итераторов рассчитан на *итерируемое значение*, но мы можем предоставить *итератор* просто потому, что итератор сам по себе является итерируемым значением! При создании экземпляра итератора по существующему итератору возвращается сам итератор.

Кроме простого использования встроенных итераторов вы также можете позаботиться о том, чтобы ваши

собственные структуры данных соответствовали протоколу итераторов; это означает, что вы сознательно утверждаете возможность потребления ваших данных циклами `for..of` и оператором `...`. Стандартизация этого протокола означает, что код становится в целом более узнаваемым и удобочитаемым.

Замыкания

Почти каждый разработчик JS пользовался в своей работе *замыканиями* (closures), даже если он этого не осознавал. Собственно, замыкания являются одним из самых распространенных видов функциональности программирования в самых разных языках. Они играют настолько фундаментальную роль, что понимать их не менее важно, чем переменные или циклы.

Замыкания могут показаться чем-то экзотическим и даже волшебным. Их обсуждения часто выглядят очень абстрактно или слишком неформально, что совершенно не помогает понять, о чем же именно идет речь.

Вы должны уметь распознавать использование замыканий в программах, так как наличие или отсутствие замыкания иногда становится причиной ошибок (и даже скрытых проблем с быстродействием).

Итак, попробуем дать прагматичное и конкретное определение замыканию.

- Замыкание возникает тогда, когда функция запоминает и продолжает обращаться к переменным, находящимся вне ее области видимости.

В этом определении необходимо обратить внимание на две характеристики. Во-первых, замыкание является частью природы функций. У объектов не бывает замыканий, у функций они могут быть. Во-вторых, чтобы наблюдать замыкание, функция должна выполняться не в той области видимости, в которой эта функция была изначально определена.

Пример:

```
function greeting(msg) {
    return function who(name) {
        console.log(`${ msg }, ${ name }!`);
    };
}

var hello = greeting("Hello");
var howdy = greeting("Howdy");

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

hello("Sarah");
// Hello, Sarah!

howdy("Grant");
// Howdy, Grant!
```

Сначала выполняется внешняя функция `greeting(..)`, создающая экземпляр внутренней функции `who(..)`; эта функция замыкается по переменной `msg`, которая является параметром из внешней области видимости `greeting(..)`. При возвращении этой внутренней функции ссылка на нее присваивается переменной `hello` из внешней области видимости. Затем `greeting(..)` вызывается во второй раз с созданием нового экземпляра внутренней функции, с новым замыканием по новой переменной `msg`, и эта ссылка возвращается для присваивания `howdy`.

Когда функция `greeting(..)` завершает выполнение, обычно мы ожидаем, что все ее переменные будут удалены из памяти. Вроде бы все переменные `msg` должны исчезнуть, но этого не происходит. Все дело в замыкании. Так как экземпляры внутренней функции все еще живы (они присвоены `hello` и `howdy` соответственно), их замыкания все еще сохраняют переменные `msg`. Эти замыкания не являются «моментальным снимком» значения переменной `msg`; это прямые ссылки, сохраняющие саму переменную. Это означает, что замыкания могут наблюдать обновления этих переменных (и вносить их) со временем.

```
function counter(step = 1) {
    var count = 0;
    return function increaseCount(){
        count = count + step;
        return count;
    };
```

```
}

var incBy1 = counter(1);
var incBy3 = counter(3);

incBy1(); // 1
incBy1(); // 2

incBy3(); // 3
incBy3(); // 6
incBy3(); // 9
```

Каждый экземпляр внутренней функции `increaseCount()` замыкается по переменным `count` и `step` из области видимости их внешней функции `counter(..)`. Переменная `step` не изменяется с течением времени, но переменная `count` обновляется при каждом вызове этой внутренней функции. Так как замыкание распространяется на переменные, а не на «моментальные снимки» этих значений, эти обновления сохраняются.

Замыкания чаще всего встречаются при работе с асинхронным кодом — например, обратными вызовами. Пример:

```
function getSomeData(url) {
    ajax(url,function onResponse(resp){
        console.log(
            `Response (from ${ url }): ${ resp }`
        );
    });
}
```

```
getSomeData("https://some.url/wherever");
// Response (from https://some.url/wherever): ...
```

Внутренняя функция `onResponse(..)` замыкается по `url` и таким образом сохраняет и запоминает его до того момента, когда вызов Ajax вернет управление и выполнит `onResponse(..)`. И хотя `getSomeData(..)` завершается немедленно, переменная-параметр `url` «живет» в замыкании столько, сколько потребуется.

Внешняя область видимости не обязана быть функцией — обычно она ей является, но не всегда. Важно лишь то, чтобы во внешней области видимости была как минимум одна переменная, к которой происходит обращение из внутренней функции.

```
for (let [idx,btn] of buttons.entries()) {
    btn.addEventListener("click",function onClick(){
        console.log(`Clicked on button (${ idx })!`);
    });
}
```

Так как в цикле используются объявления `let`, каждый итератор получает новые переменные `idx` и `btn` с блоковой (локальной) областью видимости; цикл также каждый раз создает новую внутреннюю функцию `onClick(..)`.

Внутренняя функция замыкается по переменной `idx`, сохраняя ее на то время, пока обработчик остается установленным для `btn`. Таким образом, при щелчке на каждой кнопке ее обработчик может вывести свя-

занное значение индекса, потому что обработчик запоминает свою соответствующую переменную `idx`. Не забывайте: замыкание распространяется не на значение (например, 1 или 3), а на саму переменную `idx`.

Замыкание — один из самых распространенных и важных паттернов программирования в любом языке. Но это особенно справедливо для JS; трудно представить, как сделать что-нибудь полезное без использования замыканий тем или иным образом.

Если вы все еще не чувствуете уверенности с замыканиями, большая часть книги 2, «Области видимости и замыкания», посвящена этой теме.

Ключевое слово this

Один из самых мощных механизмов JS также становится одним из самых неверно понимаемых. Речь идет о ключевом слове `this`. Одно распространенное ошибочное представление — что `this` для функции ссылается на саму функцию. Из-за того, как этот механизм работает в других языках, встречается и другое заблуждение: что `this` указывает на экземпляр, которому принадлежит метод. Оба представления ошибочны.

Как упоминалось ранее, при определении функция *присоединяется* к своей вмещающей области види-

мости через замыкание. Область видимости представляет собой набор правил, управляющих разрешением ссылок на переменные. Но кроме области видимости функции также имеют другую характеристику, которая влияет на то, к чему они могут обращаться. Эту характеристику лучше всего описать как *контекст выполнения*, и для обращения к ней из функции используется ключевое слово `this`.

Область видимости статична, и она содержит фиксированный набор переменных, доступных в конкретный момент и в точке, в которой определяется функция. *Контекст* выполнения функции динамичен, и он полностью зависит от того, **как вызывается функция** (независимо от того, где она определена и даже откуда вызывается).

`this` не является фиксированной характеристикой функции, основанной на определении функции; скорее это динамическая характеристика, которая определяется заново при каждом вызове функции.

Одна из возможных точек зрения на *контекст выполнения* — что это реально существующий объект, свойства которого доступны функции во время ее выполнения. Сравните с областью видимости, которую тоже можно рассматривать как *объект*; с другой стороны, *объект области видимости* скрыт внутри ядра JS, для функции он всегда остается одним и тем же, а его *свойства* принимают форму пере-

менных-идентификаторов, доступных внутри функции.

```
function classroom(teacher) {
    return function study() {
        console.log(
            `${ teacher } says to study ${ this.topic }`
        );
    };
}
var assignment = classroom("Kyle");
```

Внешняя функция `classroom(..)` не содержит обращения по ключевому слову `this`, поэтому эта функция ничем не отличается от всех остальных функций, которые приводились выше. Но внутренняя функция `study()` содержит ссылку на `this`, что делает ее `this`-зависимой функции. Другими словами, эта функция зависит от своего контекста выполнения.

`study()` также замыкается по переменной `teacher` из ее внешней области видимости.

Внутренняя функция `study()`, возвращаемая вызовом `classroom("Kyle")`, присваивается переменной с именем `assignment`. Как же будет вызываться `assignment()` (т. е. `study()`)?

```
assignment();
// Kyle says to study undefined -- Oops :(
```

В этом фрагменте `assignment()` вызывается как простая, обычная функция, без предоставления какого-либо контекста выполнения. Так как программа не выполняется в строгом режиме (см. главу 1, раздел «Строго говоря»), контекстно зависимые функции, вызываемые без указания какого-либо контекста, по умолчанию используют в качестве контекста глобальный объект (`window` в браузере). А так как глобальной переменной с именем `topic` не существует (и у глобального объекта нет такого свойства), `this.topic` преобразуется в `undefined`.

Пример:

```
var homework = {
topic: "JS",
assignment: assignment
};

homework.assignment();
// Kyle says to study JS
```

Копия ссылки на функцию `assignment` назначается как свойство объекта `homework`, после чего она вызывается выражением `homework.assignment()`. Это означает, что `this` для этого вызова функции будет указывать на объект `homework`. Следовательно, `this.topic` преобразуется в `"JS"`.

И наконец:

```
var otherHomework = {
    topic: "Math"
};

assignment.call(otherHomework);
// Kyle says to study Math
```

Третий способ вызова функции — метод `call(..)`, который получает объект (`otherHomework` в данном случае), используемый для назначения ссылки `this` для вызова функции. Свойство `this.topic` преобразуется в `"Math"`.

Одна контекстно зависимая функция, вызванная тремя разными способами, будет выдавать разные ответы относительно того, на какой объект указывает ссылка `this`.

Преимуществом `this`-зависимых функций — и их динамического контекста — является их способность более гибко повторно использовать одну функцию с данными из разных объектов. Функция, которая замыкается по области видимости, не может ссылаться на другую область видимости или набор переменных. Но функция с динамической контекстной зависимостью `this` может быть достаточно полезной для некоторых задач.

Прототипы

Если `this` является характеристикой выполнения функции, прототип является характеристикой объекта и конкретного преобразования обращения к свойству.

Прототип можно рассматривать как связь между двумя объектами; эта связь остается незаметной, хотя ее можно выявить и наблюдать. Связывание прототипа происходит при создании объекта; он связывается с другим, уже существующим объектом.

Серия объектов, связанных через их прототипы, называется «цепочкой прототипов».

Цель связывания прототипов (т. е. от объекта B к объекту A) заключается в том, чтобы обращения к свойствам/методам объекта B, отсутствующим в B, *делегировались* A для обработки. Делегирование обращений к свойствам/методам позволяет двум (и более!) объектам сотрудничать друг с другом для решения некоторой задачи.

Рассмотрим определение объекта в виде обычного литерала:

```
var homework = {
    topic: "JS"
};
```

Объект `homework` содержит только одно свойство: `topic`. Однако его связывание прототипа по умолчанию ведет к объекту `Object.prototype`, который содержит встроенные методы, в том числе `toString()` и `valueOf()` среди прочих.

Делегирование через связывание прототипов можно наблюдать на примере связи между `homework` и `Object.prototype`:

```
homework.toString(); // [object Object]
```

Вызов `homework.toString()` работает, хотя в `homework` не определен метод `toString()`; в результате делегирования вместо него вызывается метод `Object.prototype.toString()`.

Связывание объектов

Чтобы определить связывание с прототипом объекта, можно создать объект служебным методом `Object.create(..)`:

```
var homework = {
    topic: "JS"
};

var otherHomework = Object.create(homework);

otherHomework.topic; // "JS"
```

Первый аргумент `Object.create(..)` задает объект, с которым должен быть связан вновь создаваемый объект, после чего возвращает созданный (и связанный!) объект.

На рис. 4 показано, как три объекта (`otherHomework`, `homework` и `Object.prototype`) связываются через цепочку прототипов:

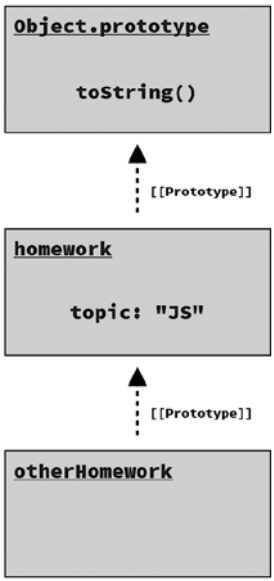


Рис. 4. Объекты в цепочке прототипов

Делегирование через цепочку прототипов применяется только для обращений, запрашивающих значение свойства. Если вы назначите свойство объекта, это

присваивание будет применяться напрямую к объекту независимо от того, с каким объектом он связан через прототип.

`Object.create(null)` создает объект, который не связывается с прототипом и поэтому является чисто автономным; в некоторых обстоятельствах такой способ может оказаться предпочтительным.

Пример:

```
homework.topic;
// "JS"

otherHomework.topic;
// "JS"

otherHomework.topic = "Math";
otherHomework.topic;
// "Math"

homework.topic;
// "JS" -- не "Math"
```

Присваивание `topic` создает свойство с этим именем непосредственно в `otherHomework`; оно никак не отражается на свойстве `topic` из `homework`. Затем следующая команда обращается к `otherHomework.topic`, и мы получаем неделегированный ответ из этого нового свойства: `"Math"`.

На рис. 5 изображены объекты/свойства после присваивания, создающего свойство `otherHomework.topic`.

Свойство `topic` в `otherHomework` замещает одноименное свойство объекта `homework` в цепочке.

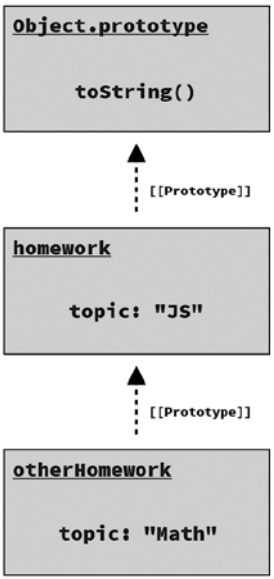


Рис. 5. Замещенное свойство topic

Другой способ создания объекта со связыванием прототипа — откровенно говоря, более запутанный, но более распространенный — основан на использовании паттерна «Прототипический класс», который существовал еще до добавления механизма class (см. главу 2, подраздел «Классы») в ES6. Эта тема более подробно рассматривается в приложении А, раздел «Прототипические классы».

Снова о this

Ключевое слово `this` уже рассматривалось выше, но по-настоящему его важность проявляется тогда, когда вы начинаете рассматривать вызовы функций с делегированием прототипам. Собственно, одна из главных причин, по которым `this` поддерживает динамический контекст, основанный на том, как была вызвана функция, заключается в том, что вызовы методов для объектов, делегируемые по цепочке прототипов, продолжают поддерживать ожидаемое значение `this`.

Пример:

```
var homework = {
    study() {
        console.log(`Please study ${ this.topic }`);
    }
};

var jsHomework = Object.create(homework);
jsHomework.topic = "JS";
jsHomework.study();
// Please study JS

var mathHomework = Object.create(homework);
mathHomework.topic = "Math";
mathHomework.study();
// Please study Math
```

Каждый из двух объектов `jsHomework` и `mathHomework` связывается по прототипу с одним объектом `homework`,

содержащим функцию `study()`. Каждый из объектов `jsHomework` и `mathHomework` содержит отдельное свойство `topic` (рис. 6).

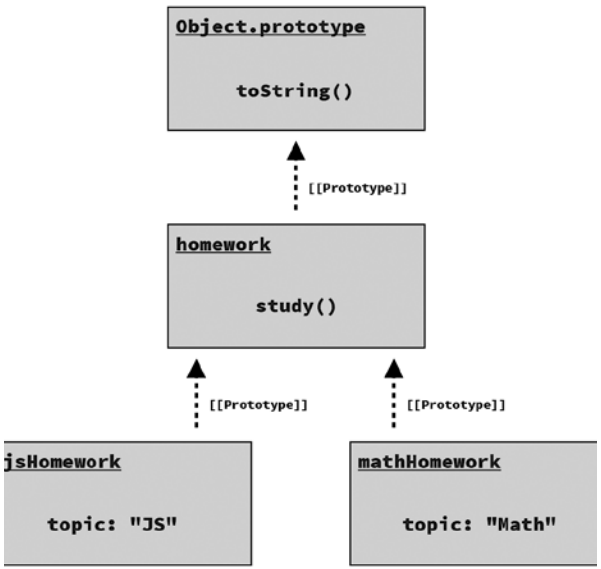


Рис. 6. Два объекта, связанных с общим родителем

`jsHomework.study()` делегирует `homework.study()`, но его ссылка `this` (`this.topic`) для этого выполнения преобразуется в `jsHomework` из-за вызова функции, так что `this.topic` преобразуется в `"JS"`. Аналогичным образом выполняется преобразование для вызова `mathHomework.study()`, делегирующего `homework.study()`, но при этом `this` все равно преобразуется

в `mathHomework`, а следовательно, `this.topic` преобразуется в `"Math"`.

Приведенный фрагмент кода был бы намного менее полезным, если бы ссылка `this` преобразовывалась в `homework`. Впрочем, во многих других языках это будет именно `homework`, потому что метод `study()` определяется для `homework`.

В отличие от многих других языков, динамическая природа `this` в JS является критическим фактором, благодаря которому делегирование прототипам — и даже `class` — работает так, как ожидалось!

А теперь «почему?»

Из этой главы прежде всего следует, что во внутренней реализации JS происходит гораздо больше, чем кажется на первый взгляд.

Когда вы начнете изучать JS более внимательно, одними из самых важных навыков, которые следует тренировать и развивать, становятся любознательность и умение задавать вопрос «почему?», когда вы сталкиваетесь с чем-то в языке.

И хотя в этой главе некоторые темы изложены достаточно глубоко, многие подробности были полностью опущены. Вам предстоит еще многое узнать, и путь к этим знаниям начинается с правильных во-

просов к вашему коду. Умение задавать правильные вопросы — важнейший навык для повышения квалификации разработчика.

В последней главе этой книги мы кратко поговорим об основных частях JS, которые рассматриваются в остальных книгах серии. Также не пропускайте приложение Б — в нем приведен код для повторения некоторых тем, рассмотренных в книге.

4 Общая картина

В этой книге приведен обзор тем, в которых необходимо разобраться *в начале изучения* JS. Книга призвана заполнить пробелы, которые могли подвести читателей, не имеющих опыта JS, на первых порах знакомства с языком. Я также надеюсь, что книга намекает на более глубокие подробности, которые разожгут ваше любопытство и побудят больше узнать о языке.

В остальных книгах серии прочие части языка рассматриваются очень детально, что было бы невозможно сделать в нескольких коротких главах.

И все же помните: не нужно торопиться. Не неситесь к следующей книге, пытаясь выхватить что-нибудь полезное на бегу, не жалейте времени и вернитесь к уже просмотренному материалу. Выделите еще немного времени на анализ кода ваших текущих проектов и сравните его с тем, что рассматривалось в книге.

В последней главе структура языка JS разделена на три столпа. Далее приводится краткий план того, чего можно ожидать от других книг серии и как бы я рекомендовал действовать дальше. Также не пропускайте приложения, особенно приложение Б — раздел «Практика, практика, практика!»

Столп 1: области видимости и замыкания

Организация переменных по областям видимости (функции, блоки) — одна из фундаментальных характеристик любого языка; возможно, никакая другая характеристика не оказывает столь значительного влияния на поведение программ.

Области видимости можно сравнить с банками, а переменные — с цветными камешками, которые вы раскладываете по этим банкам. Тогда модель области видимости языка напоминает набор правил, которые помогают определить, в какой банке должны находиться камешки конкретного цвета.

Области видимости могут вкладываться друг в друга, и для любого заданного выражения или команды доступны только переменные на этом или на одном из более высоких уровней (внешние области видимости); переменные на более низких уровнях (внутренние области видимости) скрыты и недоступны.

Так ведут себя области видимости в большинстве языков (так называемые *лексические* области видимости). Границы единиц областей видимости и принадлежность переменных к этим областям определяются во время разбора (компиляции) программы. Иначе говоря, это решение, принимаемое на стадии

создания программы: расположение функций/областей видимости в программе определяет структуру областей видимости в этой части программы.

JS использует лексические области видимости, хотя многие разработчики утверждают, что это не так, из-за двух специфических характеристик модели, отсутствующих в других языках с лексическими областями видимости.

Первая особенность обычно называется *поднятием* (hoisting): все переменные, объявленные в любой точке области видимости, интерпретируются так, словно объявлены в ее начале. Вторая особенность заключается в том, что переменные, объявленные с ключевым словом `var`, имеют функциональную область видимости, даже если располагаются в блоке.

Ни поднятия, ни функциональной области видимости `var` недостаточно для того, чтобы подкрепить утверждение об отсутствии лексической видимости в JS. У объявлений `let/const` существует специфическое ошибочное поведение, называемое временной мертвой зоной (TDZ, Temporal Dead Zone), из-за которого могут появляться наблюдаемые переменные, которые не могут использоваться. Как бы странно ни выглядела TDZ, она *также* отменяет зону лексической видимости. Все это лишь уникальные особенности языка, которые должен узнать и понять каждый разработчик JS.

Замыкание является естественным результатом лексической видимости в языках, в которых функции являются полноправными значениями, как в JS. Когда функция создает ссылки на переменные из внешней области видимости, а потом эта функция передается как значение и выполняется в других областях видимости, она сохраняет доступ к переменным исходной области видимости; в этом состоит суть замыкания.

Во всем программировании, но особенно в JS, замыкания лежат в основе многих важных паттернов проектирования, включая модули. На мой взгляд, модули лучше всего соответствуют принципам организации кода в JS.

В книге 2, «Области видимости и замыкания», вы сможете больше узнать об областях видимости, замыканиях и работе модулей.

Столп 2: прототипы

Вторым столпом языка является система прототипов. Эта тема подробно рассматривалась в главе 3, раздел «Прототипы», но хочу просто сделать несколько замечаний относительно ее важности. JS — один из очень немногих языков, в которых объекты можно создавать явно и непосредственно, без предварительного определения их структуры в классе.

Многие годы разработчики реализовывали на основе прототипов паттерн проектирования «класс», так называемое прототипическое наследование (см. приложение А, раздел «Прототипические «классы»). Затем с появлением в ES6 ключевого слова `class` язык ускорил свое движение по направлению к программированию в стиле ОО/классов.

Но я считаю, что это стремление только скрывает красоту и мощь системы прототипов — способности двух объектов просто соединиться друг с другом и взаимодействовать динамически (во время выполнения функции/метода) посредством совместного использования контекста `this`.

Классы — всего лишь один из паттернов, которые можно реализовать на основе этой мощи. Можно подойти с совершенно другой стороны: просто принять объекты как объекты, забыть о классах и предоставить объектам взаимодействовать через цепочку прототипов. Такой подход называется *делегированием поведения*. Я считаю, что для организации поведения и данных делегирование работает лучше, чем наследование классов.

Однако почти все внимание достается наследованию классов. А его остатки приходятся на долю функционального программирования (FP, functional programming) как своего рода «антиклассовому» подходу к проектированию программ. Меня это огорчает,

потому что в результате никто не рассматривает делегирование как жизнеспособную альтернативу.

Рекомендую выделить больше времени на книгу 3, «Объекты и классы», — она показывает, что делегирование объектов имеет существенно больший потенциал, чем вы, возможно, представляли себе. Я вовсе не являюсь противником классов, но намеренно выдвигаю тезис: «Классы — не единственный механизм работы с объектами». Мне хотелось бы, чтобы как можно больше разработчиков JS задумалось над этим.

На мой взгляд, делегирование объектов намного лучше соответствует духу и стилистике JS, чем классы.

Столп 3: типы и преобразования

Безусловно, третий столп JS чаще всего упускают из виду при рассмотрении природы JS.

Абсолютное большинство разработчиков совершенно превратно представляют себе работу типов в языках программирования, особенно в JS. Волна интереса в широком сообществе JS способствовала переходу на решения со статической типизацией и инструменты с поддержкой типов вроде TypeScript или Flow.

Согласен, разработчики JS должны больше знать о типах и о том, как JS управляет преобразованиями типов. Я также согласен с тем, что инструменты с поддержкой типов могут пригодиться — при условии, что разработчики уже получили эти знания и воспользовались ими!

Но я совершенно не согласен с постоянно встречающимся выводом о том, что механизм типов JS плох и типы JS необходимо прикрыть решениями, лежащими за пределами языка. Чтобы умно и основательно пользоваться типами в программе, совершенно не обязательно следовать принципам статической типизации. Есть и другие варианты, если в вас живет дух противоречий и вы хотите действовать в стиле JS (а впрочем, об этом позднее).

Пожалуй, этот столп важнее первых двух в том смысле, что ни одна программа JS не сможет сделать ничего полезного, если не будет правильно пользоваться типами значений JS, а также преобразованиями значений между разными типами.

Даже если вы любите TypeScript/Flow, то не сможете извлечь максимум пользы из них или методологий программирования без глубокого знания того, как сам язык обходится с типами значений.

Чтобы больше узнать о типах JS и преобразованиях, см. книгу «Типы и грамматические конструк-

ции»[1]. Но пожалуйста, не пропускайте эту тему только потому, что вы постоянно слышали, будто всегда должны использовать `===` и забыть обо всем остальном.

Без изучения этого столпа вы не сможете в полной мере владеть JS.

По ветру

Дам совет относительно того, как вам продолжать свое познавательное путешествие по JS и ваш путь по остальным книгам серии: помните о *ветре*.

Для начала задумайтесь над тем, как большинство людей подходит к изучению и использованию JS. Вероятно, вы уже заметили, что эти книги во многих отношениях идут против ветра. В серии YDKJSY я достаточно уважаю читателя, чтобы объяснять все части JS, не ограничиваясь популярными темами. Я искренне верю, что вы заслуживаете (и способны) получить полное представление о языке.

Но такой подход отличается от того, который применяется в большинстве существующих материалов.

---

[1] *Симпсон К.* {Вы не знаете JS} Типы и грамматические конструкции. — СПб.: Питер, 2020. — 240 с.: ил.— *Примеч. ред.*

Также он означает, что чем больше вы будете следовать рекомендациям в этих книгах — т. е. будете тщательно обдумывать и самостоятельно анализировать, что лучше подойдет для вашего кода, — тем больше вы будете выделяться среди коллег. Это может быть и хорошо, и плохо. Лучший способ выделиться на общем фоне — делать что-то по-другому.

Но многие люди также говорили мне, что приводили объяснения из книг во время собеседования и интервьюер говорил кандидату, что тот ошибается; более того, у нескольких человек, по их утверждениям, даже были отозваны офферы.

По возможности я стараюсь приводить абсолютно точную информацию о JS, источником которой обычно является сама спецификация. Но в книгах есть и мое личное мнение насчет наилучших способов интерпретации и использования JS. Я не выдаю свои мнения за факты или наоборот. Читателям всегда понятно, что есть что.

Факты, касающиеся JS, обычно неоспоримы. В спецификации либо что-то сказано, либо нет. Если вам не нравится, что говорится в спецификации, — обсуждайте это с TC39! Если на собеседовании интервьюер утверждает, что вы ошибаетесь относительно фактов, спросите, может ли он подтвердить свое высказывание по спецификации. Если интервьюер не изменит сво-

его решения, скорее всего, работать в этом месте не стоит.

Но если вы решили разделить мою точку зрения на те или иные вопросы, будьте готовы к тому, что ее придется отстаивать. Не ограничивайтесь простым повторением моих слов. Сформируйте свое мнение. Защищайте его. И если кто-то, на кого вы хотите работать, с вами не согласен, уходите. JS обширен, и в нем хватит места для множества разных подходов.

Иначе говоря, не бойтесь идти *против ветра*, как это делаю я в своих книгах и учебных курсах. Никто не сможет вам сказать, как использовать JS наиболее эффективно; решение остается за вами. Я всего лишь стараюсь дать вам возможность прийти к собственным заключениям, какими бы они ни были.

С другой стороны, существует «ветер», на который действительно следует обращать внимание и следовать за ним: это то, как работает JS на уровне языка. В JS некоторые вещи работают хорошо и естественно (при должной практике и старании), но есть и другие, которые даже не стоит пытаться делать.

Можно ли сделать так, чтобы программа на JS была внешне похожа на программу Java, C# и Perl? А как насчет Python, или Ruby, или даже PHP? В той или иной степени — безусловно, возможно. Но стоит ли?

На мой взгляд, не стоит. Считаю, что вам следует изучить и принять подход JS и писать свои программы в стиле JS настолько, насколько это практично. Кто-то скажет, что это подразумевает неформальное и неряшливое программирование, но я о другом. Просто в JS существует множество паттернов и идиом, которые легко узнаются как характерные для JS, и умение идти по ветру — стандартный путь к успеху.

Наконец, возможно самое важное, что необходимо понять: как ваши программы и коллеги-разработчики справляются со своими задачами. Не думайте, что после чтения этих книг вам удастся за ночь изменить все тенденции в существующих проектах. Такой подход всегда обречен на неудачу.

Изменения придется вносить понемногу и неспешно. Постарайтесь объяснить коллегам, почему важно снова вернуться и переосмыслить ваш подход. Но делайте это постепенно, и пусть сравнения кода «до и после» говорят за вас. Соберите всю команду для обсуждения и выступайте за решения, основанные на анализе и подтверждающей информации из кода, а не на том, что «наш старший разработчик всегда делал это так».

И это самый важный совет, который поможет вам в изучении JS. Всегда ищите возможности лучше использовать те средства, которые вам предоставляет

JS, для написания более понятного кода. Каждый, кто будет работать над вашим кодом (включая вас самого в будущем), будет вам благодарен!

По порядку

Итак, теперь у вас имеется более широкое представление о том, что вам предстоит исследовать в JS, и правильный настрой, с которым вы отправитесь в свое путешествие.

Один из самых распространенных вопросов, которые мне обычно задают в этот момент, звучит так: «А в каком порядке нужно читать книги?» И на него обычно можно дать прямолинейный ответ... но не всегда.

Большинству читателей я рекомендую осваивать книги серии в следующем порядке:

1. Начните с изучения основ JS в книге «Познакомьтесь, JavaScript» (книга 1) — хорошие новости, вы уже почти закончили эту книгу!
2. В книге «Области видимости и замыкания» (книга 2) ознакомьтесь с первым столпом JS: лексической видимостью, а также тем, как она поддерживает замыкания и как паттерн «Модуль» используется для организации кода.

3. В книге «Замыкания и объекты»[1] (книга 3) сосредоточьтесь на втором столпе JS: как работает `this` в JS, как прототипы объектов поддерживают делегирование и как прототипы делают возможным механизм классов для организации кода в ОО-стиле.

4. В книге «Типы и грамматические конструкции» (книга 4) рассматривается третий и последний столп JS: типы и преобразования типов, а также влияние синтаксиса и грамматики JS на то, как мы пишем свой код.

5. Когда вы успешно освоите все три столпа, в книге «Асинхронная обработка и оптимизация»[2] (книга 5) изучите использование программной логики для моделирования изменений состояния в программах как синхронно (немедленно), так и асинхронно (со временем).

6. Серия завершается книгой «ES6 и не только»[3] (книга 6) — взглядом в ближнее и далекое будущее JS, включая разнообразные возможности, которые

---

[1] *Симпсон К.* {Вы не знаете JS} Замыкания и объекты. — СПб.: Питер, 2020. — 336 с.: ил. — *Примеч. ред.*

[2] *Симпсон К.* {Вы не знаете JS} Асинхронная обработка и оптимизация. — СПб.: Питер, 2021. — 352 с. — *Примеч. ред.*

[3] *Симпсон К.* ES6 и не только. — СПб.: Питер, 2018. — 336 с.: ил. — *Примеч. ред.*

в скором времени с большой вероятностью появятся в программах JS.

Так выглядит рекомендуемый порядок чтения книг серии.

Однако книги 2–4 обычно можно читать в произвольном порядке в зависимости от того, какая тема вас интересует и что вам будет проще изучать сначала. Но не рекомендую пропускать хотя бы одну из этих трех книг — даже книгу «Типы и грамматика», хотя такое искушение наверняка возникнет у некоторых из вас! — даже если вы думаете, что в этой теме вы уже разбираетесь.

Книга 5 («Асинхронная обработка и оптимизация») критична для глубокого понимания JS. Но если вы начнете изучать ее и решите, что материал слишком сложный, отложите ее до тех пор, пока вы не начнете лучше разбираться в языке. Чем больше кода JS вы написали, тем больше вы оцените эту книгу. Не бойтесь возвращаться к ней в будущем.

Последняя книга серии «ES6 и не только» в некоторой степени обособлена. Ее можно читать в последнюю очередь или же сразу же после книги «Познакомьтесь, JavaScript», если вы хотите расширить свои представления о сути JS. Скорее всего, эту книгу я буду еще обновлять, так что стоит время от времени возвращаться к ней.

Но в каком бы порядке вы ни решили знакомиться с книгами YDKJSY, для начала прочитайте приложения, особенно фрагменты из приложения Б «Практика, практика, практика!». Я ведь уже говорил, что вы непременно должны практиковаться? Лучший способ научиться программировать — писать код.

A Дальнейшее изучение

В этом приложении некоторые темы из текста основных глав будут рассмотрены более подробно. Этот материал можно рассматривать как необязательный обзор некоторых тонких нюансов, встречающихся в книгах этой серии.

Значения и ссылки

В главе 2 были представлены две разновидности значений: примитивы и объекты. Однако мы еще не обсудили одно ключевое отличие между ними: то, как эти значения присваиваются и передаются при вызове.

Во многих языках разработчик может выбрать между присваиванием/передачей значения в виде самого значения или ссылки на это значение. Однако в JS это решение полностью определяется разновидностью значения. Это удивляет многих разработчиков с опытом работы на других языках, начинающих писать код на JS.

Если вы присваиваете/передаете само значение, то это значение копируется. Пример:

```
var myName = "Kyle";

var yourName = myName;
```

Здесь переменная `yourName` содержит отдельную копию строки `"Kyle"` из значения, хранящегося в `myName`. Это объясняется тем, что значение является примитивом, а примитивные значения всегда присваиваются/передаются в виде копий значений.

А вот как можно доказать, что в программе действительно существуют две отдельные копии:

```
var myName = "Kyle";

var yourName = myName;

myName = "Frank";

console.log(myName);
// Frank

console.log(yourName);
// Kyle
```

Как видите, повторное присваивание `myName` значения `"Frank"` не отразилось на `yourName`. Дело в том, что каждая переменная содержит собственную копию значения.

С другой стороны, ссылки воплощают идею о том, что две и более переменные указывают на одно и то же значение, так что изменение этого общего значения отразится на обращении по любой из этих ссылок. В JS только объектные значения (массивы, объекты, функции и т. д.) рассматриваются как ссылки.

Пример:

```
var myAddress = {
    street: "123 JS Blvd",
    city: "Austin",
    state: "TX"
};

var yourAddress = myAddress;

// Переезжаем по новому адресу!
myAddress.street = "456 TS Ave";

console.log(yourAddress.street);
// 456 TS Ave
```

Так как значение, присваиваемое `myAddress`, является объектом, оно хранится/присваивается по ссылке; таким образом, при присваивании переменной `yourAddress` копируется ссылка, а не само объектное значение. Вот почему обновленное значение, присвоенное `myAddress.street`, отражается при обращении к `yourAddress.street`. `myAddress` и `yourAddress` содержат копии ссылки на один общий объект, так что обновление значения по одной ссылке означает обновление по обеим ссылкам.

И снова JS выбирает поведение копирования значения или копирования ссылки в зависимости от типа значения. Примитивы хранятся по значению, а объекты — по ссылке. Переопределить это поведение в JS невозможно ни в том ни в другом направлении.

Многоликие функции

Вспомните следующий фрагмент из раздела «Функции» главы 2:

```
var awesomeFunction = function(coolThings) {
    // ..
    return amazingStuff;
};
```

Это функциональное выражение называется *анонимным функциональным выражением*, потому что между ключевым словом `function` и списком параметров `(..)` отсутствует идентификатор. Этот момент сбивает с толку многих разработчиков JS, потому что в ES6 JS выполняет для анонимных функций автоматическое определение имени:

```
awesomeFunction.name;
// "awesomeFunction"
```

Свойство `name` функции предоставляет либо непосредственно заданное имя (в случае объявления), либо автоматически определенное имя в случае анонимного функционального выражения. Обычно это значение используется средствами разработчика при анализе значения-функции либо при выводе трассировки стека.

Таким образом, даже анонимное функциональное выражение может получить имя. Однако автомати-

ческое определение имени выполняется только в ограниченных случаях — например, при присваивании функционального выражения (с использованием `=`). Например, если функциональное выражение передается при вызове функции, автоматическое определение имени не выполняется; свойство `name` вернет пустую строку, а на консоли разработчика обычно будет выводиться (anonymous function).

Даже при автоматическом определении имени **функция все равно остается анонимной**. Почему? Потому что автоматически определенное имя является строковым значением метаданных, а не идентификатором, который ссылается на функцию. У анонимной функции нет идентификатора, который мог бы использоваться для обращения к ней из нее самой (например, для рекурсии, отмены связывания событий и т. д.). Сравните форму анонимного функционального выражения со следующей формой:

```
// let awesomeFunction = ..
// const awesomeFunction = ..
var awesomeFunction = function someName(coolThings) {
    // ..
    return amazingStuff;
};

awesomeFunction.name;
// "someName"
```

Это функциональное выражение является *именованным функциональным выражением*, потому что иден-

тификатор `someName` напрямую связывается с функциональным выражением во время компиляции; связь с идентификатором `awesomeFunction` не происходит до достижения этой команды во время выполнения. Два идентификатора не обязаны совпадать; в одних случаях есть смысл сделать их разными, в других будет лучше, если они совпадают. Также обратите внимание на то, что явное имя функции (идентификатор `someName`) имеет более высокий приоритет при присваивании *имени* свойству `name`.

Должны ли функциональные выражения быть именованными или анонимными? Мнения по этому поводу сильно различаются. Многие разработчики предпочитают анонимные функции. Они короче и, безусловно, более распространены в существующем коде JS.

На мой взгляд, если в программе существует функция, у нее есть цель; в противном случае ее следует убрать! А если у нее есть цель, то есть и естественное имя, которое эту цель описывает.

Если у функции есть имя, то вы как автор кода должны включить это имя в код, чтобы читателю не приходилось изобретать его по написанному коду и выполнять в голове исходный код. Даже тривиальное тело функции, такое как `x * 2`, придется прочитать, чтобы изобрести для него имя вида `double` или `multBy2`; даже эта небольшая умственная работа становится

излишней, когда просто можно потратить лишнюю секунду, чтобы один раз присвоить функции имя `double` или `multBy2`. Тем самым читатели избавятся от необходимости повторять эту мыслительную работу каждый раз, когда код будет читаться в будущем.

На начало 2020-х годов в JS существовало много других форм определений функций (в некоторых отношениях это огорчает). Может, будущем появятся и другие!

Еще несколько форм объявлений:

```
// Объявление функции-генератора
function *two() { .. }

// Объявление асинхронной функции
async function three() { .. }

// Объявление асинхронной функции-генератора
async function *four() { .. }

// Объявление экспортируемой именованной функции
// (модули ES6)
export function five() { .. }
```

И еще несколько из (многих) форм функциональных выражений:

```
// IIFE
(function(){ .. })();
(function namedIIFE(){ .. })();

// asynchronous IIFE
```

```
(async function(){ .. })();
(async function namedAIIFE(){ .. })();

// arrow function expressions
var f;
f = () => 42;
f = x => x * 2;
f = (x) => x * 2;
f = (x,y) => x * y;
f = x => ({ x: x * 2 });
f = x => { return x * 2; };
f = async x => {
    var y = await doSomethingAsync(x);
    return y * 2;
};
someOperation( x => x * 2 );
// ..
```

Учтите, что стрелочные функциональные выражения являются **синтаксически анонимными** (это означает, что синтаксис не дает возможности назначить идентификатор для функции). Функциональное выражение может иметь автоматически определяемое имя, но только если оно используется в одной из форм присваивания, а не в форме (более распространенной) передачи в аргументе при вызове функции (как в последней строке фрагмента).

Так как я не считаю, что анонимные выражения стоит часто использовать в программах, я не любитель формы стрелочных функций `=>`. Такие функции обычно имеют конкретное предназначение (т. е. интерпретируют ключевое слово `this` лексически), но это не

означает, что их следует использовать для всех функций, которые вы пишете. Выбирайте самый подходящий инструмент для каждой задачи.

Функции также могут задаваться в определениях классов и определениях объектных литералов. В этих формах они обычно называются методами, хотя в JS этот термин ничем принципиально не отличается от «функции».

```
class SomethingKindaGreat {
    // class methods
    coolMethod() { .. } // без запятых!
    boringMethod() { .. }
}

var EntirelyDifferent = {
    // object methods
    coolMethod() { .. }, // с запятыми!
    boringMethod() { .. },

    // свойство с (анонимным) функциональным
    // выражением
    oldSchool: function() { .. }
};
```

Уф! Способов определения функций действительно много.

Упрощенного удобного решения не существует. Придется привыкнуть ко всем формам функций, чтобы распознавать их в существующем коде и правильно использовать в своем коде. Внимательно изучите их и почаще практикуйтесь!

Условное сравнение с преобразованием типа

Да, название получилось довольно длинным. Но о чем идет речь? Об условных выражениях, необходимых для выполнения сравнений, которые ориентируются на преобразования типов для принятия решений.

Команды `if` и `?:`-тернарные команды, а также проверочные условия в циклах `while` и `for` выполняют неявное сравнение значений. Но какого рода? Является ли оно строгим или с преобразованием типа? На самом деле и то и другое.

Пример:

```
var x = 1;

if (x) {
    // будет выполнено!
}

while (x) {
    // будет выполнено один раз!
    x = false;
}
```

Эти условные выражения можно интерпретировать так:

```
var x = 1;

if (x == true) {
    // будет выполнено!
}
```

```
while (x == true) {
    // будет выполнено один раз!
    x = false;
}
```

В этом конкретном случае — когда значение `x` равно 1 — эта модель работает, но в более общем случае она теряет точность. Пример:

```
var x = "hello";

if (x) {
    // будет выполнено!
}

if (x == true) {
    // не будет выполнено :(
}
```

Нехорошо. Что же в действительности происходит в команде `if`? Более точная мысленная модель выглядит так:

```
var x = "hello";

if (Boolean(x) == true) {
   // будет выполнено
}

// which is the same as:

if (Boolean(x) === true) {
    // будет выполнено
}
```

Так как функция `Boolean(..)` всегда возвращает значение типа `boolean`, выбор между `==` и `===` в этом фрагменте роли не играет; оба оператора делают одно и то же. Но важно понять, что перед сравнением происходит преобразование от того типа, к которому относится `x` в настоящий момент, к типу `boolean`.

Просто избавиться от преобразований типа в сравнениях JS не удастся. Соберитесь с силами и изучите их.

Прототипические классы

В главе 3 были представлены прототипы, и вы узнали, как объекты связываются через цепочку прототипов.

Другой способ связывания прототипов послужил (откровенно говоря, весьма уродливым) предшественником элегантной системы классов ES6 (см. главу 2, подраздел «Классы») — это так называемые *прототипические классы*.

Хотя этот стиль программирования довольно редко встречается в JS в наши дни, вопросы о нем невероятно часто встречаются на собеседованиях. Изучите его.

Для начала вспомним стиль программирования `Object.create(..)`:

```
var Classroom = {
welcome() {
    console.log("Welcome, students!");
}
    };

var mathClass = Object.create(Classroom);

mathClass.welcome();
// Welcome, students!
```

Здесь объект `mathClass` связывается через прототип с объектом `Classroom`. Благодаря этой связи вызов функции `mathClass.welcome()` делегируется методу, определенному в `Classroom`.

Паттерн «Прототипический класс» отнес бы поведение делегирования к наследованию и мог бы определить его (с таким же поведением) следующим образом:

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

Classroom.prototype.welcome = function hello() {
    console.log("Welcome, students!");
};

var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

Все функции по умолчанию хранят ссылку на пустой объект в свойстве с именем `prototype`. Название создает путаницу: это не *прототип* функции (с которым функция связывается прототипической связью), а объект-прототип, с которым должна устанавливаться *связь* при создании других объектов вызовом функции с `new`.

Мы добавляем в этот пустой объект (`Classroom.prototype`) свойство `welcome`, указывающее на функцию `hello()`.

Затем вызов `new Classroom()` создает новый объект (присваиваемый `mathClass`) и устанавливает его прототипическую связь с существующим объектом `Classroom.prototype`.

Хотя `mathClass` не содержит свойства-функции `welcome()`, обращение успешно делегируется функции `Classroom.prototype.welcome()`.

В настоящее время паттерн «прототипический класс» настоятельно не рекомендуется использовать. Ему на смену пришел механизм классов ES6:

```
class Classroom {
    constructor() {
        // ..
    }

    welcome() {
        console.log("Welcome, students!");
    }
}
```

```
var mathClass = new Classroom();

mathClass.welcome();
// Welcome, students!
```

Во внутренней реализации создается та же прототипическая связь, но этот синтаксис классов укладывается в рамки классово ориентированного паттерна проектирования намного четче, чем «прототипические классы».

Б Практика, практика, практика!

В этом приложении рассматриваются некоторые упражнения с предлагаемыми решениями. Они помогут потренироваться в применении концепций, описанных в книге.

Сравнения

Потренируемся в работе с типами значений и сравнениями (глава 4, столп 3), в которых должны быть задействованы преобразования типов.

Функция `scheduleMeeting(..)` должна получать время начала встречи (строка «чч:мм» в 24-часовом формате) и ее продолжительность (в минутах). Функция должна вернуть `true`, если встреча приходится полностью на рабочий день (в соответствии с временем, заданным в `dayStart` и `dayEnd`); если встреча выходит за рамки рабочего дня, возвращается `false`.

```
const dayStart = "07:30";
const dayEnd = "17:45";

function scheduleMeeting(startTime,durationMinutes) {
    // ..TODO..
}

scheduleMeeting("7:00",15);  // false
scheduleMeeting("07:15",30); // false
scheduleMeeting("7:30",30);  // true
scheduleMeeting("11:30",60); // true
scheduleMeeting("17:00",45); // true
```

```
scheduleMeeting("17:30",30); // false
scheduleMeeting("18:00",15); // false
```

Попробуйте сначала решить задачу самостоятельно. Рассмотрите возможность использования операторов проверки равенства и операторов относительного сравнения, а также учтите влияние преобразования типов на этот код. Когда у вас появится работающий код, *сравните* свое(-и) решение(-я) с кодом из раздела «Предлагаемые решения» в конце приложения.

Замыкания

Теперь попрактикуемся в использовании замыканий (глава 4, столп 1).

Функция `range(..)` получает в первом аргументе число, представляющее первое числовое значение в диапазоне. Второй аргумент также является числом, представляющим вторую границу диапазона (включительно). Если второй аргумент опущен, должна возвращаться другая функция, которая рассчитывает получить этот аргумент.

```
function range(start,end) {
    // ..TODO..
}

range(3,3); // [3]
range(3,8); // [3,4,5,6,7,8]
range(3,0); // []
```

```
var start3 = range(3);
var start4 = range(4);

start3(3); // [3]
start3(8); // [3,4,5,6,7,8]
start3(0); // []

start4(6); // [4,5,6]
```

Попробуйте сначала решить задачу самостоятельно.

Когда у вас появится работающий код, *сравните* свое решение с кодом из раздела «Предлагаемые решения» в конце приложения.

Прототипы

Наконец, потренируемся в использовании `this` и объектов, связанных через прототип (глава 4, столп 2).

Определите в программе модель «однорукого бандита» с тремя колесами, которые могут вращаться по отдельности вызовом `spin()`, а затем выводит текущие значения всех колес вызовом `display()`.

Базовое поведение одного колеса определяется приведенным ниже объектом `reel`. Однако игровому автомату потребуются отдельные объекты `reel`, которые делегируют обращения `reel`, и каждый из этих объектов должен иметь свойство `position`.

Объект `reel` умеет только выводить свой текущий символ на колесе вызовом `display()`, но «однорукие бандиты» обычно выводят по три символа на колесо: текущую позицию (`position`), позицию выше (`position - 1`) и позицию ниже (`position + 1`). Таким образом, при выводе «однорукого бандита» должна выводиться сетка символов 3 x 3.

```
function randMax(max) {
    return Math.trunc(1E9 * Math.random()) % max;
}

var reel = {
    symbols: [
        "X", "Y", "Z", "W", "$", "*", "<", "@"
    ],
    spin() {
        if (this.position == null) {
            this.position = randMax(
                this.symbols.length - 1
            );
        }
        this.position = (
            this.position + 100 + randMax(100)
        ) % this.symbols.length;
    },
    display() {
        if (this.position == null) {
            this.position = randMax(
                this.symbols.length - 1
            );
        }
        return this.symbols[this.position];
    }
};
```

```
var slotMachine = {
    reels: [
        // потребуются 3 отдельных объекта reel
        // подсказка: Object.create(..)
    ],
    spin() {
        this.reels.forEach(function spinReel(reel){
            reel.spin();
        });
    },
    display() {
        // TODO
    }
};

slotMachine.spin();
slotMachine.display();
// < | @ | *
// @ | X | <
// X | Y | @

slotMachine.spin();
slotMachine.display();
// Z | X | W
// W | Y | $
// $ | Z | *
```

Попробуйте сначала решить задачу самостоятельно.

Подсказки

- Используйте оператор остатка от деления `%` для циклического возврата `position` к началу при обращении к символам на колесе.

- Используйте `Object.create(..)` для создания объекта и связывания через прототип с другим объектом. После установления связи делегирование позволит объектам совместно использовать контекст `this` во время вызова метода.
- Вместо того чтобы изменять объект `reel` напрямую для вывода трех значений `position`, можно использовать другой временный объект (снова `Object.create(..)`) со своим значением `position`, от которого исходит делегирование.

Когда у вас появится работающий код, *сравните* свое решение с кодом из раздела «Предлагаемые решения» в конце приложения.

Предлагаемые решения

Учтите, что здесь предлагается лишь один из вариантов решения. У этих упражнений есть много потенциальных решений. Сравните свой подход с приведенным, проанализируйте достоинства и недостатки каждого варианта.

Предлагаемое решение для упражнения «Сравнения» (столп 3):

```
const dayStart = "07:30";
const dayEnd = "17:45";
```

```
function scheduleMeeting(startTime,durationMinutes) {
    var [ , meetingStartHour, meetingStartMinutes ] =
        startTime.match(/^(\d{1,2}):(\d{2})$/) || [];

    durationMinutes = Number(durationMinutes);

    if (
        typeof meetingStartHour == "string" &&
        typeof meetingStartMinutes == "string"
    ) {
        let durationHours =
            Math.floor(durationMinutes / 60);
        durationMinutes =
            durationMinutes - (durationHours * 60);
        let meetingEndHour =
            Number(meetingStartHour) + durationHours;
        let meetingEndMinutes =
            Number(meetingStartMinutes) +
            durationMinutes;

        if (meetingEndMinutes >= 60) {
            meetingEndHour = meetingEndHour + 1;
            meetingEndMinutes =
                meetingEndMinutes - 60;
        }

        // сформировать полностью уточненные строки
        // времени (для упрощения сравнений)
        let meetingStart = `${
            meetingStartHour.padStart(2,"0")
        }:${
            meetingStartMinutes.padStart(2,"0")
        }`;
        let meetingEnd = `${
            String(meetingEndHour).padStart(2,"0")
        }:${
            String(meetingEndMinutes).padStart(2,"0")
```

```
        }`;

        // ПРИМЕЧАНИЕ: так как все выражения являются
        // строками, все сравнения являются
        // алфавитными, однако здесь такое сравнение
        // безопасно, потому что используются
        // полностью уточненные строки времени
        // (например, "07:15" < "07:30")
        return (
            meetingStart >= dayStart &&
            meetingEnd <= dayEnd
        );
    }

    return false;
}

scheduleMeeting("7:00",15);  // false
scheduleMeeting("07:15",30); // false
scheduleMeeting("7:30",30);  // true
scheduleMeeting("11:30",60); // true
scheduleMeeting("17:00",45); // true
scheduleMeeting("17:30",30); // false
scheduleMeeting("18:00",15); // false
```

Предлагаемое решение для упражнения «Замыкания» (столп 1):

```
function range(start,end) {
    start = Number(start) || 0;

    if (end === undefined) {
        return function getEnd(end) {
            return getRange(start,end);
        };
    }
```

```
    else {
        end = Number(end) || 0;
        return getRange(start,end);
    }

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

    function getRange(start,end) {
        var ret = [];
        for (let i = start; i <= end; i++) {
            ret.push(i);
        }
        return ret;
    }
}

range(3,3); // [3]
range(3,8); // [3,4,5,6,7,8]
range(3,0); // []

var start3 = range(3);
var start4 = range(4);

start3(3); // [3]
start3(8); // [3,4,5,6,7,8]
start3(0); // []

start4(6); // [4,5,6]
```

Предлагаемое решение для упражнения «Прототипы» (столп 2):

```
function randMax(max) {
    return Math.trunc(1E9 * Math.random()) % max;
}
```

```
var reel = {
    symbols: [
        "X", "Y", "Z", "W", "$", "*", "<", "@"
    ],
    spin() {
        if (this.position == null) {
            this.position = randMax(
                this.symbols.length - 1
            );
        }
        this.position = (
            this.position + 100 + randMax(100)
        ) % this.symbols.length;
    },

    display() {
        if (this.position == null) {
            this.position = randMax(
                this.symbols.length - 1
            );
        }
        return this.symbols[this.position];
    }
};

var slotMachine = {
    reels: [
        Object.create(reel),
        Object.create(reel),
        Object.create(reel)
    ],
    spin() {
        this.reels.forEach(function spinReel(reel){
            reel.spin();
        });
    },
    display() {
```

```
        var lines = [];

        // display all 3 lines on the slot machine
        for (
            let linePos = -1; linePos <= 1; linePos++
        ) {
            let line = this.reels.map(
                function getSlot(reel){
                    var slot = Object.create(reel);
                    slot.position = (
                        reel.symbols.length +
                        reel.position +
                        linePos
                    ) % reel.symbols.length;
                    return reel.display.call(slot);
                }
            );
            lines.push(line.join(" | "));
        }

        return lines.join("\n");
    }
};

slotMachine.spin();
slotMachine.display();
// < | @ | *
// @ | X | <
// X | Y | @

slotMachine.spin();
slotMachine.display();
// Z | X | W
// W | Y | $
// $ | Z | *
```

Вот и все, что я хотел рассказать в этой книге! Теперь пришло время поискать реальные проекты, на которых можно потренироваться в применении этих идей. Просто продолжайте программировать, ведь это лучший способ чему-то научиться!

Кайл Симпсон

{Вы пока еще не знаете JS} Познакомьтесь, JavaScript. 2-е изд.

Перевел с английского Е. Матвеев

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

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.
Дата изготовления: 08.2021.
Наименование: книжная продукция.
Срок годности: не ограничен.
Налоговая льгота — общероссийский классификатор продукции ОК 034-2014,
58.11.12 — Книги печатные профессиональные, технические и научные. Импортер
в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск,
ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.
Подписано в печать 13.08.21. Формат 60x90/16. Бумага офсетная. Усл. п. л. 12,000.
Тираж 1000. Заказ

{Вы пока еще не знаете JS}
Область видимости и замыкания.
2-е международное издание

Кайл Симпсон

КУПИТЬ

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

Профессиональный TypeScript. Разработка масштабируемых JavaScript-приложений

Борис Черный

КУПИТЬ

Любой программист, работающий с языком с динамической типизацией, подтвердит, что задача масштабирования кода невероятно сложна и требует большой команды инженеров. Вот почему Facebook, Google и Microsoft придумали статическую типизацию для динамически типизированного кода. Работая с любым языком программирования, мы отслеживаем исключения и вычитываем код строку за строкой в поиске неисправности и способа ее устранения. TypeScript позволяет автоматизировать эту неприятную часть процесса разработки. TypeScript, в отличие от множества других типизированных языков, ориентирован на прикладные задачи. Он вводит новые концепции, позволяющие выражать идеи более кратко и точно и легко создавать масштабируемые и безопасные современные приложения.

JavaScript для профессиональных веб-разработчиков. 4-е международное издание

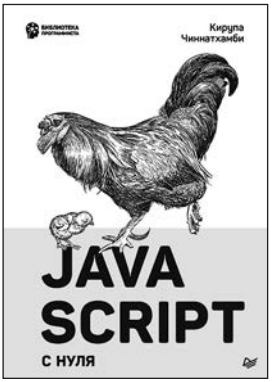
М. Фрисби

КУПИТЬ

Самое полное руководство по современному JavaScript Как максимально прокачать свои навыки и стать топовым JS-программистом? Четвертое издание «JavaScript для профессиональных веб-разработчиков» идеально подойдет тем, кто уже имеет базовые знания и опыт разработки на JavaScript. Автор сразу переходит к техническим деталям, которые сделают ваш код чистым и переведут вас с уровня рядового кодера на высоту продвинутого разработчика. Рост мобильного трафика увеличивает потребность в адаптивном динамическом веб-дизайне, а изменения в JS-движках происходят постоянно, так что каждый веб-разработчик должен постоянно обновлять свои навыки работы с JavaScript.

JavaScript с нуля

Кирупа Чиннатхамби



JavaScript еще никогда не был так прост! Вы узнаете все возможности языка программирования без общих фраз и неясных терминов. Подробные примеры, иллюстрации и схемы будут понятны даже новичку. Легкая подача информации и живой юмор автора превратят нудное заучивание в занимательную практику по написанию кода. Дойдя до последней главы, вы настолько прокачаете свои навыки, что сможете решить практически любую задачу, будь то простое перемещение элементов на странице или даже собственная браузерная игра.

КУПИТЬ