

O'REILLY®

«Превосходное введение в фундаментальные
средства JavaScript, о которых вам
никогда не расскажут (да и не смогут рассказать)
в описаниях инструментариев языка».

— Дэвид Уолш, старший веб-разработчик, Mozilla

КАЙЛ СИМПСОН

ТИПЫ &

ГРАММАТИЧЕСКИЕ

КОНСТРУКЦИИ

ВЫ НЕ ЗНАЕТЕ

S





Types & Grammar

Kyle Simpson

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®



ТИПЫ &
ГРАММАТИЧЕСКИЕ
КОНСТРУКЦИИ

КАЙЛ СИМПСОН



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

2019

ББК 32.988.02-018

УДК 004.738.5

С37

Симпсон К.

С37 {Вы не знаете JS} Типы и грамматические конструкции. — СПб.: Питер, 2019. — 240 с.: ил. — (Серия «Бестселлеры O'Reilly»).

ISBN 978-5-4461-1266-1

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное руководство исследует типы более глубоко, чем все существующие книги: вы узнаете, как работают типы, о проблемах их преобразования и научитесь пользоваться новыми возможностями.

Как и в других книгах серии «Вы не знаете JS», здесь показаны нетривиальные аспекты языка, от которых программисты JavaScript предпочитают держаться по-дальше (или полагают, что они не существуют). Вооружившись этими знаниями, вы достигнете истинного мастерства JavaScript.

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

ББК 32.988.02-018

УДК 004.738.5

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

Информация, содержащаяся в данной книге, получена из источников, рассматриваемых издательством как надежные. Тем не менее, имея в виду возможные человеческие или технические ошибки, издательство не может гарантировать абсолютную точность и полноту приводимых сведений и не несет ответственности за возможные ошибки, связанные с использованием книги. Издательство не несет ответственности за доступность материалов, ссылки на которые вы можете найти в этой книге. На момент подготовки книги к изданию все ссылки на интернет-ресурсы были действующими.

ISBN 978-1491904190 англ.

Authorized Russian translation of the English edition of You Don't Know JS: Types & Grammar (ISBN 9781491904190)
© 2015 Getify Solutions, Inc.

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

© Перевод на русский язык ООО Издательство «Питер», 2019

© Издание на русском языке, оформление
ООО Издательство «Питер», 2019

© Серия «Бестселлеры O'Reilly», 2019

ISBN 978-5-4461-1266-1

Оглавление

Предисловие	9
Введение	11
Задача	12
О книге.	14
Типографские соглашения	14
Использование программного кода примеров	15
От издательства	16
Глава 1. Типы	17
Хоть типом назови его, хоть нет.	18
Встроенные типы	19
Значения как типы	22
<code>undefined</code> и необъявленные переменные	23
<code>typeof</code> для необъявленных переменных	24
Итоги	28
Глава 2. Значения	30
Массивы	30
Подобие массивов	32
Строки	33
Числа	37
Синтаксис работы с числами	37
Малые дробные значения	42
Безопасные целочисленные диапазоны	44

Проверка целых чисел	45
32-разрядные целые числа (со знаком)	46
Специальные значения	46
Пустые значения	47
Undefined	47
Специальные числа	50
Специальное равенство	57
Значения и ссылки	58
Итоги	64
Глава 3. Встроенные объекты (natives)	66
Внутреннее свойство [[Class]]	68
Упаковка	69
Ловушки при работе с объектными обертками	70
Распаковка	71
Встроенные объекты как конструкторы	72
Array(..)	72
Object(..), Function(..) и RegExp(..)	77
Date(..) и Error(..)	79
Symbol(..)	81
Встроенные прототипы	82
Итоги	86
Глава 4. Преобразование типов	87
Преобразование значений	87
Абстрактные операции	90
ToString	90
ToNumber	97
ToBoolean	99
Явное преобразование типов	104
Явные преобразования: String <--> Number	105

Явные преобразования: разбор числовых строк	115
Явные преобразования: * --> Boolean	120
Неявное преобразование	122
Неявное упрощение	124
Неявные преобразования: String <--> Number	125
Неявные преобразования: Boolean --> Number	130
Неявные преобразования: * --> Boolean	132
Операторы и &&	134
Преобразование символических имен	139
Равенство строгое и нестрогое	140
Быстродействие проверки равенства	141
Абстрактная проверка равенства	142
Особые случаи	151
Абстрактное относительное сравнение	162
Итоги	165
Глава 5. Грамматика	166
Команды и выражения	167
Завершающие значения команд	168
Побочные эффекты выражений	171
Правила контекста	177
Приоритет операторов	186
Ускоренная обработка	190
Плотное связывание	191
Ассоциативность	192
Неоднозначности	196
Автоматические точки с запятой	198
Исправление ошибок	200
Ошибки	202
Преждевременное использование переменных	204
Аргументы функций	205

try..finally	208
switch	212
Итоги	215
Приложение А. JavaScript в разных средах	218
Дополнение В (ECMAScript)	218
Web ECMAScript	219
Управляющие объекты	221
Глобальные переменные DOM	222
Встроенные прототипы	223
Прокладки совместимости (shims)/полифили (polyfills)	227
<script>ы	229
Зарезервированные слова	233
Ограничения реализации	234
Итоги	235
Об авторе	236

Предисловие

Есть хорошая поговорка: «JavaScript – единственный язык, которым разработчики пользуются, не научившись им пользоваться».

Каждый раз, слыша эту фразу, я смеюсь, потому что она оказалась совершенно справедливой для меня и, как подозреваю, для многих других разработчиков. JavaScript (а возможно, даже CSS и HTML) не входил в число основных языков, которые преподавались в колледжах в начале эпохи интернета. Таким образом, личностное развитие во многом зависело от умения начинающего разработчика искать информацию и просматривать разметку, для того чтобы разобраться в базовых веб-языках.

Я до сих пор помню свой первый проект веб-сайта в средней школе. Я должен был построить сайт рандомного веб-магазина. Будучи фанатом Джеймса Бонда, я решил создать магазин на тему «Золотого глаза». В нем было все: MIDI-тема из фильма, игравшая в фоновом режиме; курсор-прицел на базе JavaScript; звук выстрела, воспроизведившийся при каждом щелчке. Агент Кью был бы горд подобным мастерством.

Зачем я это рассказываю? Потому что тогда я сделал то, что сегодня делают многие разработчики: я скопировал фрагменты кода JavaScript в мой проект, не имея ни малейшего представления о том, что же в действительности происходит. Массовое использование инструментариев JavaScript (таких, как jQuery) в каком-то отношении способствует этой тенденции – нежеланию изучать базовый язык JavaScript.

Я вовсе не предлагаю отказаться от инструментариев JavaScript; в конце концов, я вхожу в JavaScript-команду разработки MooTools! Но инструментарии JavaScript стали такими мощными именно благодаря тому, что их разработчики знали основные возможности и связанные с ними ловушки и прекрасно применяли их на практике. Какими бы полезными ни были эти инструментарии, чрезвычайно важно знать основы языка. А с такими книгами, как серия «Вы не знаете JS» Кайла Симпсона, уже ничего не оправдывает нежелание изучать их.

«Типы и грамматические конструкции» дает представление об основных возможностях JavaScript, которым вас никогда не научат копирование и инструментарии JavaScript. Преобразование типов и его проблемы, встроенные объекты (natives) как конструкторы и весь спектр основ JavaScript тщательно объясняются на специально подобранных примерах. Как и в других книгах серии, Кайл сразу переходит к сути без лишних разговоров и «воды», я предпочитаю именно такие технические книги.

Читайте и держите книгу поближе к своему рабочему столу!

*Дэвид Уолш (David Walsh) (<http://davidwalsh.name>),
старший веб-разработчик Mozilla*

Введение

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

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

Так как JavaScript заимствует концепции и синтаксические идиомы из нескольких языков, включая процедурные корни в стиле C и менее очевидные функциональные корни в стиле Scheme/Lisp, он в высшей степени доступен для широкого спектра разработчиков — даже обладающих минимальным опытом программи-

рования. Программа «Hello World» на JavaScript настолько проста, что язык располагает к себе и кажется удобным с самых первых минут знакомства.

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

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

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

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

Задача

Если каждый раз, сталкиваясь с каким-то сюрпризом или неприятностью в JavaScript, вы заносите их в «черный список» (как многие привыкли делать), вскоре от всей моши JavaScript у вас

останется лишь скорлупа. Хотя это подмножество принято называть «Хорошими Частями», я призываю вас, дорогой читатель, рассматривать его как «Простые Части», «Безопасные Части» и даже «Неполные Части».

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

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

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

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

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

О книге

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



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

Типографские соглашения

В этой книге приняты следующие типографские соглашения:

Курсив

Используется для обозначения новых терминов.

Моноширинный шрифт

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

Моноширинный курсив

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



Так выделяются советы и предложения.



Так обозначаются советы, предложения и примечания общего характера.



Так обозначаются предупреждения и предостережения.

Использование программного кода примеров

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

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

Мы приветствуем, но не требуем добавлять ссылку на первоисточник при цитировании. Под ссылкой на первоисточник мы подразумеваем указание авторов, издательства и ISBN.

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 Типы

Многие разработчики скажут, что в динамическом языке (таком, как JS) вообще нет *типов*. Посмотрим, что на этот счет сказано в спецификации ES5.1¹:

«Алгоритмы из этой спецификации работают со значениями, с каждым из которых связан тип. Возможные значения типов точно определены в этой секции. Типы дополнительно классифицируются на типы языка ECMAScript и типы спецификации.

Тип языка ECMAScript соответствует значениям, с которыми программист ECMAScript работает напрямую с использованием языка ECMAScript. Типы языка ECMAScript — `undefined`, `null`, `boolean`, `string`, `number` и `object`».

Возможно, поклонники языков с сильной (статической) типизацией будут возражать против такого использования слова «тип». В этих языках термин «тип» означает гораздо *больше*, чем в JS.

Кто-то скажет, что JS не должен претендовать на наличие «типов». Правильнее было бы называть эти конструкции «тегами» или, скажем, «подтипами».

¹ <http://www.ecma-international.org/ecma-262/5.1/>.

Вот еще! Мы будем использовать следующее приближенное определение (которое, похоже, было заложено в формулировку из спецификации): *тип* представляет собой внутренний встроенный набор характеристик, которые однозначно определяют поведение конкретного значения и отличают его от других значений – как для движка, так и для разработчика.

Иначе говоря, если и движок, и разработчик интерпретируют значение 42 (число) не так, как они интерпретируют значение "42" (строка), то эти два значения имеют разные *типы* – `number` и `string` соответственно. Используя 42, вы *хотите* выполнить некую численную операцию – например, математические вычисления. Но при использовании "42" вы *намереваетесь* сделать нечто, связанное со строками, вывести данные на страницу и т. д. Эти два значения относятся к разным типам.

Такое определение ни в коем случае не идеально. Тем не менее для нашего обсуждения достаточно и такого. Кроме того, оно соответствует тому, как JS описывает свое поведение.

Хоть типом назови его, хоть нет...

Если не считать расхождений в академических определениях, почему так важно, есть в JavaScript типы или нет?

Правильное понимание каждого типа и его внутреннего поведения абсолютно необходимо для понимания того, как правильно и точно преобразовывать значения к разным типам (см. главу 4). Практически любая когда-либо написанная программа JS должна преобразовывать значения в той или иной форме, поэтому очень важно, чтобы вы делали это ответственно и уверенно.

Если имеется число 42, которое вы хотите интерпретировать как строку (например, извлечь "2" как символ в позиции 1), очевидно, значение сначала необходимо преобразовать из числа в строку.

На первый взгляд кажется, что это просто.

Однако такие преобразования могут выполняться множеством разных способов. Некоторые из них выражаются явно, логически объяснимы и надежны. Но если вы будете недостаточно внимательны, преобразования могут выполняться очень странным и неожиданным образом.

Пожалуй, путаница с преобразованиями — один из главных источников огорчений для разработчиков JavaScript. Преобразования часто критиковали как настолько *опасные*, что они относились к дефекту проектирования языка, от которого стоит держаться подальше.

Вооружившись полным пониманием типов JavaScript, мы постараемся показать, почему *плохая репутация* преобразований в основном преувеличена и отчасти незаслуженна. Попытаемся поменять вашу точку зрения, чтобы вы увидели мощь и полезность преобразований. Но сначала нужно гораздо лучше разобраться со значениями и типами.

Встроенные типы

JavaScript определяет семь встроенных типов:

- `null`
- `undefined`
- `boolean`
- `number`
- `string`
- `object`
- `symbol` — добавлен в ES6!



Все эти типы, за исключением `object`, называются «примитивами».

Оператор `typeof` проверяет тип заданного значения и всегда возвращает одно из семи строковых значений — как ни странно, между ними и семью встроенными типами, перечисленными ранее, нет полностью однозначного соответствия:

```
typeof undefined      === "undefined"; // true
typeof true           === "boolean";    // true
typeof 42             === "number";     // true
typeof "42"           === "string";    // true
typeof { life: 42 }   === "object";    // true

// Добавлен в ES6!
typeof Symbol()       === "symbol";    // true
```

Шесть перечисленных типов имеют значения соответствующего типа и возвращают строковое значение с тем же именем. `Symbol` — новый тип данных, появившийся в ES6; он будет описан в главе 3.

Возможно, вы заметили, что я исключил из этого списка `null`. Это *особенный* тип. Особенный в том смысле, что его поведение с оператором `typeof` ошибочно:

```
typeof null === "object"; // true
```

Было бы удобно (и правильно!), если бы оператор возвращал `"null"`, но эта исходная ошибка в JS существует уже почти два десятилетия и, скорее всего, никогда не будет исправлена. От этого ошибочного поведения зависит столько существующего веб-контента, что «исправление» ошибки только *создаст* еще больше «ошибок» и нарушит работу многих веб-программ.

Если вы хотите проверить значение `null` по типу, вам понадобится составное условие:

```
var a = null;  
(!a && typeof a === "object"); // true
```

Null — единственное примитивное значение, которое является «ложным» (см. главу 4), но при этом возвращает "object" при проверке typeof.

Какое же седьмое строковое значение может вернуть typeof?

```
typeof function a(){ /* .. */ } === "function"; // true
```

Легко решить, что function является высокоуровневым встроенным типом в JS, особенно при таком поведении оператора typeof. Тем не менее при чтении спецификации вы увидите, что function на самом деле ближе к «подтипу» object. А именно функция называется там «вызываемым объектом» — то есть объектом с внутренним свойством [[Call]], которое позволяет активизировать его посредством вызова.

Тот факт, что функции в действительности являются объектами, имеет ряд важных следствий. Самое важное — то, что они могут обладать свойствами. Пример:

```
function a(b,c) {  
    /* .. */  
}
```

У объекта функции есть свойство length, в котором хранится количество формальных параметров в объявлении этой функции:

```
a.length; // 2
```

Так как функция была объявлена с двумя формальными именованными параметрами (b и c), «длина» функции равна 2.

Как насчет массивов? Они встроены в JS — может, им выделен отдельный тип?

```
typeof [1,2,3] === "object"; // true
```

Нет, самые обычные объекты. Наиболее уместно рассматривать массивы как «подтип» объектов (см. главу 3), в данном случае с дополнительными характеристиками числового индексирования (вместо обычных строковых ключей, как у простых объектов) и поддержания автоматически обновляемого свойства `.length`.

Значения как типы

В языке JavaScript у переменных нет типов — типы есть у *значений*. Переменная может хранить любое значение в любой момент времени.

Также на типы JS можно взглянуть под другим углом: в JS нет «принудительного контроля типов». Иначе говоря, движок не требует, чтобы в *переменной* всегда хранились переменные *исходного типа*, с которым она начала свое существование. В одной команде присваивания переменной может быть присвоена строка, в другой — число, и т. д.

Значение 42 обладает внутренним типом `number`, и этот *тип* изменить не удастся. Другое значение — например, "42" с типом `string` — может быть создано *на основе* значения 42 типа `number`, для чего используется процесс, называемый *преобразованием типов* (см. главу 4).

Применение `typeof` к переменной не означает «к какому типу относится эта переменная», как могло бы показаться на первый взгляд, потому что переменные JS не обладают типами. Вместо этого вы спрашиваете: «К какому типу относится значение в этой переменной?»

```
var a = 42;
typeof a; // "number"

a = true;
typeof a; // "boolean"
```

Оператор `typeof` всегда возвращает строку. Таким образом:

```
typeof typeof 42; // "string"
```

Первый оператор `typeof 42` возвращает `"number"`, а `typeof "number"` возвращает `"string"`.

undefined и необъявленные переменные

Переменные, *на данный момент* не имеющие значения, на самом деле имеют значение `undefined`. Вызов `typeof` для таких переменных возвращает `"undefined"`:

```
var a;  
typeof a; // "undefined"  
  
var b = 42;  
var c;  
  
// В будущем  
b = c;  
typeof b; // "undefined"  
typeof c; // "undefined"
```

Многим разработчикам удобно рассматривать слово `"undefined"` как синоним для необъявленной переменной. Тем не менее в JS эти две концепции заметно отличаются.

Неопределенная переменная (`undefined`) была объявлена в доступной области видимости, но на данный момент не содержит никакого другого значения. С другой стороны, необъявленная переменная не была формально объявлена в доступной области видимости.

Пример:

```
var a;  
  
a; // undefined  
b; // ReferenceError: значение b не определено
```

Путаница возникает из-за сообщения об ошибке, которое выдается браузерами по этому условию. Как видите, выдается сообщение «значение `b` не определено» (`b is not defined`), очень легко и даже логично спутать его с «переменная `b` содержит `undefined`». Тем не менее `undefined` и «не определено» (`not defined`) – это совершенно разные понятия. Конечно, лучше бы браузеры выдавали сообщение вида «значение `b` не найдено» или «значение `b` не объявлено» для предотвращения путаницы!

Также `typeof` проявляет для необъявленных переменных специальное поведение, которое только усиливает путаницу.

Пример:

```
var a;  
  
typeof a; // "undefined"  
  
typeof b; // "undefined"
```

Оператор `typeof` возвращает `"undefined"` даже для «необъявленных» переменных. Обратите внимание: при выполнении `typeof b` не было выдано сообщения об ошибке, хотя переменная `b` и не объявлена. Это специальная защита для `typeof`.

Как я уже говорил, было бы лучше, если бы `typeof` для необъявленных переменных возвращал специальную строку `"undeclared"`, вместо того чтобы объединять возвращаемое значение с совершенно другим случаем `"undefined"`.

typeof для необъявленных переменных

Впрочем, эта защита может быть полезной при использовании JavaScript в браузерах, где несколько разных файлов сценариев могут загружать переменные в общее глобальное пространство имен.



Многие разработчики полагают, что в глобальном пространстве имен не должно быть никаких переменных, а все переменные должны содержаться в модулях и приватных/раздельных пространствах имен. Теоретически идея выглядит прекрасно, но на практике это невозможно; однако это достойная цель, к которой следует стремиться! К счастью, в ES6 добавилась полноценная поддержка модулей, благодаря которой такое разделение со временем станет куда более реальным.

Простой пример: представьте, что в вашей программе предусмотрен «режим отладки», который включается глобальной переменной (флагом) с именем `DEBUG`. Прежде чем выводить в консоль сообщение, нужно проверить, была ли эта переменная объявлена. Высокоуровневое объявление глобальной переменной `var DEBUG = true` будет включено только в файл `debug.js`, который загружается в браузере только в ходе разработки/тестирования, но не в среде реальной эксплуатации.

Вы должны внимательно отнестись к проверке глобальной переменной `DEBUG` в остальном коде приложения, чтобы избежать ошибки `ReferenceError`. В этом случае защита `typeof` приходит на помощь:

```
// Упс! Это вызовет ошибку!
if (DEBUG) {
    console.log( "Debugging is starting" );
}

// Безопасная проверка существования
if (typeof DEBUG !== "undefined") {
    console.log( "Debugging is starting" );
}
```

Подобные проверки полезны даже в том случае, если вы не работаете с переменными, определяемыми пользователем (как в случае с `DEBUG`). Если вы проводите проверку для встроенного API, возможно, также будет полезно сделать это без выдачи ошибки:

```
if (typeof atob === "undefined") {  
    atob = function() { /*..*/ };  
}
```



Если вы определяете «полифил» (polyfill) для функции, которая еще не существует, вероятно, лучше избежать использования `var` для объявления `atob`. Если вы объявили `var atob` внутри команды `if`, это объявление «поднимается» вверх области видимости, даже если условие `if` не проходит (потому что глобальная версия `atob` уже существует!). В некоторых браузерах и для некоторых специальных типов глобальных встроенных переменных (часто называемых «объектами-хостами») этот дубликат объявления может вызвать ошибку. Отсутствие `var` предотвращает поднятие объявления.

Также существует другой способ выполнения тех же проверок глобальных переменных без защиты `typeof`: проверка того, что все глобальные переменные также являются свойствами глобального объекта, которым в браузере фактически является объект `window`. Таким образом, те же проверки можно (вполне безопасно) выполнить следующим образом:

```
if (window.DEBUG) {  
    // ..  
}  
  
if (!window.atob) {  
    // ..  
}
```

В отличие от обращений к необъявленным переменным, при попытке обратиться к несуществующему свойству объекта (даже глобального) ошибка `ReferenceError` не выдается.

С другой стороны, ручное обращение к глобальной переменной по имени `window` — практика, которой некоторые разработчики предпочитают избегать, особенно если код должен выполняться в разных средах JS (не только браузерах, но и в `node.js` на стороне

сервера, например), в которых глобальная переменная не всегда называется `window`.

С технической точки зрения защита `typeof` полезна, даже если вы не используете глобальные переменные, хотя эти обстоятельства встречаются реже, а некоторые разработчики считают это решение менее желательным. Представьте, что вы написали вспомогательную функцию, которая должна копироваться другими программистами в их программы и модули, и теперь вы хотите узнать, определила ли вызывающая программа некоторую переменную (и тогда ее можно использовать) или нет.

```
function doSomethingCool() {  
    var helper =  
        (typeof FeatureXYZ !== "undefined") ?  
        FeatureXYZ :  
        function() { /*.. действия по умолчанию ..*/ };  
  
    var val = helper();  
    // ..  
}
```

`doSomethingCool()` проверяет переменную с именем `FeatureXYZ`. Если переменная будет найдена, то она используется, а если нет — использует собственную версию. Если кто-то включит эту функцию в свой модуль/программу, то функция безопасно проверит, была определена переменная `FeatureXYZ` или нет:

```
// IIFE (см. "Немедленно вызываемые функциональные выражения"  
// в книге "Область видимости и замыкания" этой серии)  
(function(){  
    function FeatureXYZ() { /*.. моя функциональность XYZ ..*/ }  
  
    // include `doSomethingCool(..)`  
    function doSomethingCool() {  
        var helper =  
            (typeof FeatureXYZ !== "undefined") ?  
            FeatureXYZ :  
            function() { /*.. действия по умолчанию ..*/ };  
  
        var val = helper();
```

```
// ...
}

doSomethingCool();
})();
```

Здесь `FeatureXYZ` не является глобальной переменной, но мы все равно используем защиту `typeof`, чтобы обезопасить ее проверку. И что важно, здесь *нет* объекта, который можно было бы использовать для проверки (как это делалось для глобальных переменных с `window.____`), так что `typeof` здесь сильно помогает.

Другие разработчики предпочитают паттерн проектирования, называемый «внедрением зависимостей». В этом случае вместо неявной проверки определения `FeatureXYZ` где-то снаружи, `doSomethingCool()` требует явной передачи зависимости:

```
function doSomethingCool(FeatureXYZ) {
  var helper = FeatureXYZ ||
    function() { /*.. Действия по умолчанию ..*/ };

  var val = helper();
  // ..
}
```

Существует много вариантов проектирования подобной функциональности. Ни один паттерн не может быть «правильным» или «ошибочным», у каждого есть свои плюсы и минусы. Однако в целом довольно удобно, что защита `typeof` для необъявленных переменных предоставляет нам больше возможностей.

Итоги

В JavaScript есть семь встроенных *типов*: `null`, `undefined`, `boolean`, `number`, `string`, `object` и `symbol`. Тип значений можно идентифицировать оператором `typeof`.

Переменные не имеют типов, но зато типы есть у хранящихся в них значений. Типы определяют поведение, присущее этим значениям.

Многие разработчики считают, что «не определено» (`undefined`) и «не объявлено» — приблизительно одно и то же, но в JavaScript эти понятия заметно отличаются. `Undefined` — значение, которое может храниться в объявленной переменной. «Не объявлено» означает, что переменная не была объявлена.

К сожалению, JavaScript отчасти объединяет эти два термина, не только в сообщениях об ошибках («`ReferenceError: a is not defined`»), но и в возвращаемых значениях оператора `typeof`, который в обоих случаях возвращает `"undefined"`.

Тем не менее защита `typeof` (предотвращение ошибки) при использовании с необъявленной переменной бывает достаточно полезной в некоторых случаях.

2 Значения

Массивы, строки и числа — основные структурные элементы любой программы. Однако в JavaScript эти типы обладают некоторыми уникальными особенностями, которые могут как порадовать вас, так и привести в замешательство.

Давайте рассмотрим некоторые встроенные типы значений в JS и разберемся, как лучше понять и использовать особенности их поведения.

Массивы

В отличие от других языков с принудительным контролем типов, массивы JavaScript представляют собой контейнеры для значений любого типа, от `string` до `number` и `object`, и даже другого массива (именно так создаются многомерные массивы):

```
var a = [ 1, "2", [3] ];  
  
a.length;      // 3  
a[0] === 1;    // true  
a[2][0] === 3; // true
```

Вам не нужно заранее объявлять размеры массивов. Вы можете просто объявить их и добавлять значения так, как вы считаете нужным:

```
var a = [ ];
a.length; // 0
a[0] = 1;
a[1] = "2";
a[2] = [ 3 ];
a.length; // 3
```



Вызов `delete` для значения-массива удаляет из массива элемент, но даже при удалении последнего элемента свойство `length` не обновляется, так что будьте осторожны! Оператор `delete` подробно рассматривается в главе 5.

Будьте внимательны с созданием «разреженных» массивов (в которых остаются или создаются пустые/отсутствующие элементы):

```
var a = [ ];
a[0] = 1;
// Элемент `a[1]` здесь не задается
a[2] = [ 3 ];
a[1]; // undefined
a.length; // 3
```

Хотя это решение работает, оно может привести к странному поведению с остающимися «пустыми элементами». Хотя элемент вроде бы должен содержать значение `undefined`, он будет вести себя не так, как при явном присваивании (`a[1] = undefined`). За дополнительной информацией обращайтесь к разделу «`Array(..)`».

Массивы индексируются числами (как и следовало ожидать), но есть один нюанс: они также являются объектами, к которым

могут добавляться строковые ключи/свойства (не учитываемые в длине массива):

```
var a = [ ];
a[0] = 1;
a["foobar"] = 2;
a.length;      // 1
a["foobar"];   // 2
a.foobar;      // 2
```

Но здесь кроется ловушка: если строковое значение, которое должно использоваться в качестве ключа, может быть преобразовано в стандартное десятичное число, то предполагается, что вы хотите использовать его в качестве числового индекса, а не строкового ключа!

```
var a = [ ];
a["13"] = 42;
a.length; // 14
```

Вообще добавлять строковые ключи/свойства к массивам не рекомендуется. Используйте для хранения значений в ключах/свойствах объекты, а массивы оставьте исключительно для значений с числовым индексированием.

Подобие массивов

В некоторых случаях бывает нужно преобразовать подобие массива (коллекцию значений с числовым индексированием) в полноценный массив. Обычно это делается для того, чтобы для коллекции значений можно было вызывать методы массивов (`indexOf(..)`, `concat(..)`, `forEach(..)` и т. д.).

Например, разные операции получения информации модели DOM возвращают списки элементов DOM, которые не являются

ся полноценными массивами, но достаточно близки к массивам для целей преобразования. Другой типичный пример — функции, предоставляющие объект `arguments` (подобие массива, в ES6 объявлен устаревшим) для обращения к аргументам в виде списка.

Один очень распространенный способ выполнения таких преобразований основан на вызове функции `slice(..)`:

```
function foo() {  
  var arr = Array.prototype.slice.call( arguments );  
  arr.push( "bam" );  
  console.log( arr );  
}  
  
foo( "bar", "baz" ); // [ "bar", "baz", "bam" ]
```

Если `slice()` вызывается без других параметров, как это происходит в приведенном фрагменте, значения по умолчанию для параметров фактически дублируют массив (или в данном случае подобие массива).

В ES6 также появилась встроенная функция `Array.from(..)`, которая делает то же самое:

```
...  
var arr = Array.from( arguments );  
...
```



`Array.from(..)` поддерживает ряд очень полезных возможностей, которые подробно рассмотрены в книге «ES6 и не только» этой серии.

Строки

Очень многие разработчики считают, что строки (`string`) по сути являются обычными массивами с элементами-символами. Хотя во внутренней реализации могут использоваться (или не ис-

пользоваться) массивы, важно понимать, что строки JavaScript на самом деле не являются массивами символов. Сходство в лучшем случае поверхностное.

Например, возьмем следующие два значения:

```
var a = "foo";
var b = ["f", "o", "o"];
```

Строки действительно отчасти напоминают массивы — они являются массивоподобными, как и упоминавшиеся ранее объекты. Например, и строки, и массивы поддерживают свойство `length`, метод `indexOf(..)` (в ES5 только для массивов) и метод `concat(..)`:

[source,js]

```
a.length;                      // 3
b.length;                      // 3

a.indexOf( "o" );              // 1
b.indexOf( "o" );              // 1

var c = a.concat( "bar" );      // "foobar"
var d = b.concat( ["b", "a", "r"] ); // ["f", "o", "o", "b", "a", "r"]

a === c;                      // false
b === d;                      // false

a;                            // "foo"
b;                            // ["f", "o", "o"]
```

Выходит, и то и другое — «массивы символов»? Не совсем так:

```
a[1] = "0";
b[1] = "0";

a; // "foo"
b; // ["f", "0", "o"]
```

Строки JavaScript неизменяемы, тогда как массивы вполне могут изменяться (муттировать). Более того, форма обращения к символу `a[1]` не всегда является единственной в JavaScript. Старые версии IE не поддерживали этот синтаксис (хотя в новых версиях

ях эта форма поддерживается). Вместо него *приходилось использовать* `a.charAt(1)`.

Другое следствие неизменяемости строк заключается в том, что ни один строковый метод не может изменить свое содержимое «на месте», он должен создавать и возвращать новые строки. С другой стороны, многие методы массивов, изменяющие содержимое массива, *вносят* изменения «на месте»:

```
c = a.toUpperCase();
a === c;      // false
a;           // "foo"
c;           // "FOO"

b.push( "!" );
b;           // ["f", "o", "o", "!"]
```

Кроме того, многие методы массивов, которые могли бы пригодиться при работе со строками, для них недоступны, но вы можете «позаимствовать» неизменяющиеся методы массивов для строки:

```
a.join;        // undefined
a.map;         // undefined

var c = Array.prototype.join.call( a, "-" );
var d = Array.prototype.map.call( a, function(v){
    return v.toUpperCase() + ".";
} ).join( "" );

c;           // "f-o-o"
d;           // "F.O.O."
```

Возьмем другой пример: перестановка строки в обратном порядке (кстати, этот вопрос часто задается на собеседованиях по JavaScript!). У массивов имеется метод `reverse()`, вносящий изменения «на месте», но у строк такого метода нет:

```
a.reverse;    // undefined
b.reverse();   // ["!", "o", "O", "f"]
b;           // ["!", "o", "O", "f"]
```

К сожалению, «заимствование» не работает с массивами, изменяющими массивы, потому что строки неизменны, а следовательно, не могут изменяться «на месте»:

```
Array.prototype.reverse.call( a );
// все равно возвращает объектную обертку String (см. главу 3)
// для "foo" :(
```

Другое обходное решение заключается в преобразовании строки в массив, выполнении нужной операции и преобразовании обратно в строку:

```
var c = a
  // разбиение `a` на массив символов
  .split( "" )
  // переставить массив символов в обратном порядке
  .reverse()
  // снова объединить массив символов в строку
  .join( "" );
c; // "oof"
```

Выглядит некрасиво? Так оно и есть. Но такое решение *работает* для простых строк, поэтому если вам понадобится решение «на скорую руку», то такой вариант вполне подойдет.



Будьте осторожны! Такое решение не работает для строк, содержащих сложные символы (Юникод): многобайтовые символы и т. д. Для правильного выполнения таких операций потребуется более сложная библиотека. За информацией обращайтесь к работе Матиаса Байненса (Mathias Bynens): *Esrever* (<https://github.com/mathiasbynens/esrever>).

На происходящее также можно взглянуть иначе: если вы часто выполняете со «строками» операции, которые интерпретируют их как *массивы символов*, возможно, проще хранить их в виде *массивов* вместо строк. Скорее всего, вы избавитесь от постоянных хлопот с преобразованием строк в массивы. А когда вам понадо-

бится строковое представление, для массива символов всегда можно вызвать `join("")`.

Числа

В JavaScript существует всего один числовой тип: `number`. Этот тип включает как «целые» значения, так и дробные числа. Я заключаю «целые» в кавычки, потому что JS давно критиковали за то, что они не являются полноценными целыми числами, как в других языках. Возможно, когда-нибудь в будущем ситуация изменится, но пока для всего приходится использовать `number`.

Таким образом, в JS «целое» число представляет собой значение, не имеющее дробной части. Таким образом, `42.0` является таким же «целым» числом, как и `42`.

Как и во многих современных языках, включая практически все языки сценариев, реализация чисел в JavaScript базируется на стандарте IEEE 754, часто называемом «плавающей точкой». В JavaScript используется формат «двойной точности» (64-разрядное двоичное представление) стандарта.

В интернете можно найти замечательные статьи о технических подробностях хранения в памяти двоичных чисел с плавающей точкой и следствиях этих решений. Так как понимание формата двоичных данных в памяти не является строго необходимым для понимания того, как правильно пользоваться числами в JS, читатель, заинтересовавшийся этой темой, может изучить спецификацию IEEE 754 самостоятельно.

Синтаксис работы с числами

Числовые литералы обычно выражаются в JavaScript в десятичном формате с точкой:

```
var a = 42;  
var b = 42.3;
```

Если начальная часть дробного значения равна 0, ее можно опустить:

```
var a = 0.42;  
var b = .42;
```

Аналогичным образом, если завершающая (дробная) часть значения после . равна 0, она также может быть опущена:

```
var a = 42.0;  
var b = 42.;
```



Запись 42. выглядит довольно непривычно. Пожалуй, это не лучшая мысль, особенно если вы хотите избежать путаницы при чтении вашего кода другими людьми. Тем не менее такая запись допустима.

По умолчанию большинство чисел форматируется в десятичной системе счисления с удалением завершающих 0 в дробной части:

```
var a = 42.300;  
var b = 42.0;  
  
a; // 42.3  
b; // 42
```

Очень большие или очень малые числа по умолчанию форматируются в экспоненциальной форме по аналогии с выводом метода `toExponential()`:

```
var a = 5E10;  
a; // 50000000000  
a.toExponential(); // "5e+10"  
  
var b = a * a;  
b; // 2.5e+21
```

```
var c = 1 / a;  
c; // 2e-11
```

Так как числовые значения могут быть упакованы в объектную обертку `Number` (см. главу 3), для работы с ними могут использоваться методы, встроенные в `Number.prototype` (также см. главу 3). Например, метод `toFixed(..)` позволяет задать количество знаков в дробной части, которые должны использоваться для представления числа:

```
var a = 42.59;  
  
a.toFixed( 0 ); // "43"  
a.toFixed( 1 ); // "42.6"  
a.toFixed( 2 ); // "42.59"  
a.toFixed( 3 ); // "42.590"  
a.toFixed( 4 ); // "42.5900"
```

Обратите внимание на то, что вывод в действительности является строковым представлением числа, а если запросить больше знаков в дробной части, чем содержит значение, оно дополняется нулями справа.

Метод `toPrecision(..)` работает аналогично, но он определяет, сколько *значащих цифр* должно использоваться для представления значения:

```
var a = 42.59;  
  
a.toPrecision( 1 ); // "4e+1"  
a.toPrecision( 2 ); // "43"  
a.toPrecision( 3 ); // "42.6"  
a.toPrecision( 4 ); // "42.59"  
a.toPrecision( 5 ); // "42.590"  
a.toPrecision( 6 ); // "42.5900"
```

Для обращения к этим методам не обязательно использовать переменную, содержащую нужное значение; их можно вызывать прямо для литералов. Будьте осторожны с оператором `..`. Поскольку `.` является действительным символом числа, этот символ

сначала будет интерпретирован как часть числового литерала (если это возможно), а не как метод доступа к свойству:

```
// неверный синтаксис:  
42.toFixed( 3 );      // SyntaxError  
  
// все эти варианты допустимы:  
(42).toFixed( 3 );  // "42.000"  
0.42.toFixed( 3 );  // "0.420"  
42..toFixed( 3 );   // "42.000"
```

Синтаксис `42.toFixed(3)` недействителен, потому что `.` поглощается как часть литерала `42`. (который допустим — см. выше!); таким образом, для обращения `.toFixed` оператора `.` уже не останется.

Конструкция `42..toFixed(3)` работает потому, что первый символ `.` является частью числа, а второй символ `.` интерпретируется как оператор свойства. Да, выглядит она странно, и в реальном коде JavaScript нечто похожее встречается очень редко. Даже прямые обращения к методам для примитивных значений достаточно необычны, хотя «необычный» не означает «плохой» или «неправильный».



Некоторые библиотеки расширяют встроенный `Number.prototype` (см. главу 3) для поддержки дополнительных операций с числами. В таких случаях абсолютно допустимо использовать конструкции вида `10..makeItRain()` для запуска 10-секундной анимации с денежным дождем или какой-нибудь похожей ерунды.

Также технически допустим следующий вариант (обратите внимание на пробел):

```
42 .toFixed(3); // "42.000"
```

Однако именно для числовых литералов такой стиль программирования создает особенно большую путаницу и не делает ничего,

а только сбивает с толку других разработчиков (в том числе и вас в будущем). Не используйте его.

Числа также могут задаваться в экспоненциальной форме, часто используемой для представления больших чисел:

```
var onethousand = 1E3; // означает 1 * 10^3
var onemilliononehundredthousand = 1.1E6; // означает 1.1 * 10^6
```

Числовые литералы также могут записываться в других системах счисления — двоичной, восьмеричной и шестнадцатеричной.

В текущих версиях JavaScript работают следующие форматы:

```
0xf3; // шестнадцатеричная запись для 243
0xF3; // то же
```

```
0363; // восьмеричная запись для 243
```



Начиная с ES6 в режиме `+ strict`, форма `0363` для восьмеричных литералов стала недопустимой (новая форма описана внизу). Форма `0363` все еще разрешена без режима `strict`, однако вам лучше перестать пользоваться ею с расчетом на будущее (и потому, что к этому моменту вы уже должны были привыкнуть к использованию `strict`).

В ES6 также допустимы следующие новые формы:

```
0o363; // восьмеричная запись для 243
00363; // то же
```

```
0b11110011; // двоичная запись для 243
0B11110011; // то же
```

Пожалуйста, окажите услугу своим коллегам-разработчикам: никогда не используйте форму `00363`. Размещая `0` рядом с буквой `O` верхнего регистра, вы сами создаете путаницу. Всегда используйте предикаты в нижнем регистре: `0x`, `0b` и `0o`.

Малые дробные значения

Самый знаменитый побочный эффект использования двоичных чисел с плавающей точкой (который, напомню, относится ко *всем* языкам, использующим IEEE 754, а не *только* к JavaScript, как ошибочно считают многие):

```
0.1 + 0.2 === 0.3; // false
```

С математической точки зрения результат должен быть истинным. Почему он ложен?

Дело в том, что **0,1** и **0,2** в двоичном формате с плавающей точкой имеют не точное, а приближенное представление, поэтому при суммировании результат не равен в точности **0,3**. Значение получается *очень* близким (**0,3000000000000004**), но если условие не выполняется, то «близость» уже несущественна.



Стоит ли JavaScript перейти на другую реализацию с точными представлениями для всех значений? Некоторые разработчики считают именно так. За прошедшие годы было представлено много альтернатив. Ни одна из них не была принята и, скорее всего, принята не будет. Как бы просто все ни выглядело со стороны, но махнуть рукой и сказать: «Исправьте уже эту ошибку!» нельзя, на самом деле все намного сложнее, иначе проблема бы уже давно была решена.

А теперь вопрос: если мы не можем *рассчитывать* на точность некоторых чисел, означает ли это, что числа вообще нельзя использовать? Конечно, нет.

Во многих ситуациях вам придется действовать осторожно, особенно при работе с дробными значениями. Достаточно часто приходится работать только с целыми числами, и более того, только с числами не более нескольких миллионов или триллион-

нов. В таких случаях всегда было (и всегда будет) абсолютно безопасно использовать числовые операции в JS.

Но что *делать*, если вам все же нужно сравнить два числа (например, $0.1 + 0.2$ с 0.3), хотя вы знаете, что простая проверка равенства не работает?

Самое распространенное решение — использовать маленькую «погрешность округления» как *допуск* для сравнения. Это крошечное значение часто называется «машинным эпсилоном», и для чисел, используемых в JavaScript, оно обычно равно 2^{-52} ($2.220446049250313e-16$).

В ES6 `Number.EPSILON` заранее определяется с этим значением допуска, так что если вы хотите использовать его, вы можете безопасно использовать полифил для версий, предшествующих ES6:

```
if (!Number.EPSILON) {  
    Number.EPSILON = Math.pow(2, -52);  
}
```

`Number.EPSILON` может использоваться для проверки двух чисел на «равенство» (в пределах погрешности округления):

```
function numbersCloseEnoughToEqual(n1, n2) {  
    return Math.abs( n1 - n2 ) < Number.EPSILON;  
}  
  
var a = 0.1 + 0.2;  
var b = 0.3;  
  
numbersCloseEnoughToEqual( a, b ); // true  
numbersCloseEnoughToEqual( 0.0000001, 0.0000002 ); // false
```

Максимальное представимое значение с плавающей точкой равно приблизительно $1.798e+308$ (это очень, очень, очень много!); оно также заранее определено в `Number.MAX_VALUE`. На другом конце оси `Number.MIN_VALUE` равно приблизительно $5e-324$ — это значение не отрицательно, но очень близко к нулю!

Безопасные целочисленные диапазоны

Из-за особенностей представления чисел существует диапазон «безопасных» значений для «целых» чисел, и он значительно меньше `Number.MAX_VALUE`.

Максимальное целое число, которое может быть «безопасно» представлено (то есть существует гарантия того, что заданное значение действительно имеет однозначное представление), равно $2^{53} - 1$, что соответствует приблизительно 9007199254740991. Если разбить число на группы разрядов, вы увидите, что оно в действительности превышает 9 квадриллионов. В общем, число слишком большое, чтобы создать какие-либо неудобства.

В действительности это значение автоматически заранее определяется в ES6 в форме `Number.MAX_SAFE_INTEGER`. Не приходится удивляться тому, что есть и наименьшее безопасное значение -9007199254740991, и оно определяется в ES6 под именем `Number.MIN_SAFE_INTEGER`.

Самая распространенная ситуация, в которой JS-программы сталкиваются с такими большими числами – работа с 64-разрядными идентификаторами из баз данных и т. д. 64-разрядные числа не могут быть точно представлены типом `number`, поэтому они должны храниться (и передаваться JavaScript и обратно) в строковом представлении.

К счастью, числовые операции с такими большими значениями идентификаторов (кроме сравнения, которое нормально работает со строками) встречаются не так уж часто. Но если вам потребуется выполнить вычисления с этими очень большими значениями, вы пока можете воспользоваться поддержкой *больших чисел*. Возможно, в будущих версиях JavaScript большие числа получат официальную поддержку.

Проверка целых чисел

Чтобы проверить, является ли значение целым числом, можно воспользоваться методом `Number.isInteger(..)`, определенным в ES6:

```
Number.isInteger( 42 );      // true
Number.isInteger( 42.000 );  // true
Number.isInteger( 42.3 );    // false
```

Полифил `Number.isInteger(..)` для браузеров до ES6:

```
if (!Number.isInteger) {
  Number.isInteger = function(num) {
    return typeof num == "number" && num % 1 == 0;
  };
}
```

Чтобы проверить, является ли значение безопасным *целым числом*, используйте метод `Number.isSafeInteger(..)`, определенный в ES6:

```
Number.isSafeInteger( Number.MAX_SAFE_INTEGER );      // true
Number.isSafeInteger( Math.pow( 2, 53 ) );            // false
Number.isSafeInteger( Math.pow( 2, 53 ) - 1 );        // true
```

Полифил `Number.isSafeInteger(..)` для браузеров до ES6:

```
if (!Number.isSafeInteger) {
  Number.isSafeInteger = function(num) {
    return Number.isInteger( num ) &&
      Math.abs( num ) <= Number.MAX_SAFE_INTEGER;
  };
}
```

32-разрядные целые числа (со знаком)

Хотя представление целых чисел позволяет безопасно использовать диапазон приблизительно до 9 квадриллионов (53 бита), некоторые числовые операции, например, поразрядные операторы, определены только для 32-разрядных чисел, так что «безопасный диапазон» для чисел, используемых таким образом, должен быть намного меньше.

Этот диапазон включает числа от `Math.pow(-2, 31)` (-2147483648, около -2,1 миллиарда) до `Math.pow(2, 31)-1` (2147483647, около +2,1 миллиарда).

Чтобы привести числовое значение к 32-разрядному целому со знаком, используйте операцию `|0`. Такое решение подходит, потому что поразрядный оператор `|` работает только для 32-разрядных целых значений (это означает, что он берет только 32 бита, а все остальные биты будут потеряны). Объединение с `0` операцией «ИЛИ» на уровне отдельных разрядов фактически является пустой операцией.



Некоторые специальные значения, которые будут рассмотрены в следующем разделе (такие, как `Nan` и `Infinity`), не являются «безопасными для 32-разрядных операций» в том отношении, что эти значения при передаче поразрядному оператору проходят через абстрактную операцию `ToInt32` (см. главу 4) и становятся для поразрядной операции обычным значением `+0`.

Специальные значения

Среди различных типов встречаются несколько специальных значений, которые *ответственный* JS-разработчик должен знать и правильно использовать.

Пустые значения

Для типа `undefined` существует только одно значение: `undefined`. Для типа `null` существует только одно значение: `null`. Итак, в обоих случаях имя определяет как тип, так и его значение.

`Undefined` и `null` часто считают синонимами для «пустых» или «несуществующих» значений. Другие разработчики предпочитают различать их по нетривиальному признаку, например:

- `null` — пустое значение;
- `undefined` — отсутствующее значение.

Или:

- `undefined` еще не имеет значения;
- `null` когда-то имело значение, а теперь его не имеет.

Как бы вы ни решили «определить» и использовать эти два значения, `null` является специальным ключевым словом, а не идентификатором, а следовательно, не может использоваться как переменная, которой можно что-то присвоить (хотя с чего бы?). Однако `undefined` (к сожалению) является идентификатором. Вот так номер!

Undefined

Если вы не используете режим `strict`, возможно (хотя и в высшей степени нежелательно) присвоить значение глобальному идентификатору `undefined`:

```
function foo() {  
  undefined = 2; // очень плохая мысль!  
}  
  
foo();  
  
function foo() {
```

```
"use strict";
undefined = 2; // TypeError!
}

foo();
```

Однако и в режиме `strict`, и без него вы можете создать локальную переменную с именем `undefined`. Впрочем, это тоже совершенно ужасная идея!

```
function foo() {
  "use strict";
  var undefined = 2;
  console.log( undefined ); // 2
}

foo();
```

Друзья не позволяют друзьям переопределять `undefined`. Никогда.

Оператор `void`

Хотя `undefined` является встроенным идентификатором, который содержит встроенное значение `undefined` (если он не был изменен, см. выше), это значение также можно получить при помощи оператора `void`.

Выражение `void` __ «стирает» любое значение, так что результат выражения всегда является **неопределенным**. Оно не изменяет существующего значения, а всего лишь гарантирует, что операторное выражение не вернет никакого значения:

```
var a = 42;

console.log( void a, a ); // undefined 42
```

По принятому соглашению (в основном из языка С) для отдельного представления значения `undefined` при помощи `void` используется запись `void 0` (хотя понятно, что `void true` и вообще

любое выражение с `void` сделает то же самое). Между `void 0`, `void 1` и `undefined` нет никаких практических различий.

Однако оператор `void` также может пригодиться в других случаях, например, если вы хотите гарантировать, что выражение не имеет возвращаемого значения (даже если оно имеет побочные эффекты). Пример:

```
function doSomething() {  
    // примечание : `APP.ready` предоставляется приложением  
    if (!APP.ready) {  
        // попробовать позднее  
        return void setTimeout( doSomething,100 );  
    }  
  
    var result;  
  
    // заняться чем-то другим  
    return result;  
}  
  
// получилось с первой попытки?  
if (doSomething()) {  
    // заняться другими задачами  
}
```

Функция `setTimeout(..)` возвращает числовое значение (уникальный идентификатор интервального таймера на случай, если вы захотите отменить его), но мы хотим скрыть его, чтобы возвращаемое значение нашей функции не давало ложный положительный результат.

Многие разработчики предпочитают выполнять те же действия по отдельности; такой способ работает так же, но без использования оператора `void`:

```
if (!APP.ready) {  
    // попробовать позднее  
    setTimeout( doSomething,100 );  
    return;  
}
```

В общем, если в каком-то месте вашей программы существует значение (полученное в результате вычисления некоторого выражения), а вам было бы удобно, чтобы вместо этого значения было `undefined`, используйте оператор `void`. Скорее всего, такие ситуации будут относительно редко встречаться, но в этих редких случаях они могут быть полезны.

Специальные числа

Тип `number` включает несколько специальных значений. Рассмотрим каждое из них более подробно.

NaN

Любая математическая операция, у которой один из двух операндов не является числом (или значением, которое может быть интерпретировано как обычное число в десятичной или шестнадцатеричной системе), не может произвести допустимое число; вместо этого будет получено значение `NaN`.

`NaN` в действительности означает «не число» (Not A Number), хотя этот термин/описание выбран крайне неудачно, и как вы вскоре увидите, только создает путаницу. Было бы намного правильнее считать `NaN` «недействительным числом», «некорректным числом» и даже «плохим числом», нежели «не числом».

Пример:

```
var a = 2 / "foo";           // NaN
typeof a === "number"; // true
```

Иначе говоря, «не число имеет тип `number`! Премия за самые непонятные имена и семантику.

`NaN` — *сигнальное значение* (значение, нормальное в других отношениях, которому был присвоен специальный смысл), пред-

ставляющее особую разновидность состояний ошибок в числовом множестве. По сути, состояние ошибки должно означать: «Я попытался выполнить математическую операцию, ничего не получилось, и вот вам результат — некорректное число».

Итак, если переменная содержит некоторое значение, и вы хотите проверить, является ли оно специальным признаком некорректного числа `NaN`, логично попробовать напрямую сравнить его с `NaN`, как и с любым другим значением, таким как `null` или `undefined`. Ничего подобного.

```
var a = 2 / "foo";  
  
a == NaN;    // false  
a === NaN;   // false
```

`NaN` — особенное значение, потому что оно никогда не равно другому значению `NaN` (то есть никогда не равно само себе). Это единственное значение, поэтому оно не рефлексивно (и не обладает свойством тождественности `x === x`). Таким образом, `NaN != NaN`. Немного странно, вы не находите?

Как же выполнять проверку на `NaN`, если сравнивать с `NaN` нельзя (поскольку такое сравнение всегда завершается неудачей)?

```
var a = 2 / "foo";  
  
isNaN( a ); // true
```

Просто, не правда ли? Нужно использовать встроенную глобальную функцию `isNaN(..)`, и она скажет вам, является ли значение `NaN` или нет. Проблема решена!

Не так быстро.

У функции `isNaN(..)` есть фатальный недостаток. Похоже, она пытается интерпретировать `NaN` («не число») слишком буквально, то есть пытается проверить, является ли переданное значение числом или не числом. Тем не менее это не совсем точно:

```
var a = 2 / "foo";
var b = "foo";

a; // NaN
b; "foo"

window.isNaN( a ); // true
window.isNaN( b ); // true... ой!
```

Очевидно, значение "foo" буквально *не является числом*, но оно определено не является значением `NaN`! Эта ошибка присутствует в JS с самого начала (это «ой» длится уже более 19 лет).

В ES6 наконец-то появилась замена: `Number.isNaN(..)`. Простой полифил позволит безопасно проверять значения `NaN` даже в браузерах до ES6:

```
if (!Number.isNaN) {
  Number.isNaN = function(n) {
    return (
      typeof n === "number" &&
      window.isNaN( n )
    );
  };
}

var a = 2 / "foo";
var b = "foo";

Number.isNaN( a ); // true
Number.isNaN( b ); // false – теперь правильно!
```

Вообще говоря, полифил для `Number.isNaN(..)` можно реализовать еще проще – использовать тот странный факт, что значение `NaN` не равно само себе. `NaN` – *единственное* значение во всем языке, для которого выполняется это условие; все остальные значения всегда *равны сами себе*.

Итак:

```
if (!Number.isNaN) {
  Number.isNaN = function(n) {
```

```
        return n !== n;  
    };  
}
```

Выглядит странно? Но работает!

`NaN`, случайно или намеренно, используется во многих реальных программах JS. Желательно использовать надежную проверку вроде `Number.isNaN(..)` в показанном виде (или в виде полифила), чтобы правильно распознать их.

Если в настоящее время вы используете только `isNaN(..)` в программе, то сталкиваетесь с печальной реальностью: в вашей программе есть ошибка, даже если вы с ней еще не столкнулись!

Бесконечности

Разработчики с опытом традиционных компилируемых языков (таких, как C) привыкли, что попытка выполнения некоторых операций приводит к ошибке компиляции или исключению времени выполнения, например, как при делении на нуль в следующих операциях:

```
var a = 1 / 0;
```

Тем не менее в JS такая операция определена, и в результате ее выполнения будет получено значение `Infinity` (оно же `Number.POSITIVE_INFINITY`). Вполне ожидаемо:

```
var a = 1 / 0; // Infinity  
var b = -1 / 0; // -Infinity
```

Как вы видите, значение `-Infinity` (оно же `Number.NEGATIVE_INFINITY`) возникает при делении на нуль, при котором один операнд (но не оба сразу!) отрицателен.

JS использует конечные числовые представления (IEEE 754 с плавающей точкой, которое было рассмотрено ранее), поэтому

в отличие от «чистой» математики переполнение возможно даже при таких операциях, как сложение и вычитание. В этих случаях вы получите `Infinity` или `-Infinity`.

Пример:

```
var a = Number.MAX_VALUE;      // 1.7976931348623157e+308
a + a;                      // Infinity
a + Math.pow( 2, 970 );      // Infinity
a + Math.pow( 2, 969 );      // 1.7976931348623157e+308
```

Согласно спецификации, если операция (например, сложение) дает значение, слишком большое для представления в IEEE 754, результат определяется режимом округления. Если говорить упрощенно, `Number.MAX_VALUE + Math.pow(2, 969)` ближе к `Number.MAX_VALUE`, чем к `Infinity`, поэтому он выполняет «округление в меньшую сторону», тогда как `Number.MAX_VALUE + Math.pow(2, 970)` ближе к `Infinity`, поэтому выполняется «округление в большую сторону».

Если думать об этом слишком долго, начнет болеть голова. Так что не задумывайтесь. Серьезно, хватит!

После переполнения в направлении одной из *бесконечностей* возврата уже не будет. Иначе говоря, от конечного можно перейти к бесконечному, но от бесконечного к конечному уже не вернуться. Поэтично...

Почти философский вопрос: «Что получится, если бесконечность разделить на бесконечность?» Наш наивный мозг, скорее всего, ответит «1», а возможно, «бесконечность». Оказывается, оба ответа неправильны. С точки зрения математики, и JavaScript операция `Infinity / Infinity` не определена. В JS будет получен результат `NaN`.

А если разделить любое положительное конечное число на `Infinity`? Легко! Получится `0`. А если любое отрицательное конечное число разделить на `Infinity`?.. Продолжайте читать.

Нули

Хотя это может привести в замешательство читателя, сведущего в математике, в JavaScript существует обычный нуль `0` (также называемый положительным нулем `+0`) и отрицательный нуль `-0`. Прежде чем объяснять, для чего вообще существует `-0`, сначала нужно разобраться, как JS с ним работает. Кроме того, что отрицательный нуль записывается в виде литерала `-0`, он также получается в результате некоторых математических операций. Пример:

```
var a = 0 / -3; // -0
var b = 0 * -3; // -0
```

Отрицательный нуль не может быть получен в результате операций сложения и вычитания.

При анализе в консоли разработчика отрицательный нуль обычно отображается в виде `-0`, хотя это стало распространенным явлением лишь недавно, так что в некоторых старых браузерах он все еще отображается в виде `0`.

Но при попытке преобразования отрицательного нуля к строковому виду, согласно спецификации, он всегда должен отображаться в виде `"0"`:

```
var a = 0 / -3;

// консоли (некоторых браузеров) поступают правильно
a;                                // -0

// но спецификация настаивает на том, что нужно врать!
a.toString();                      // "0"
a + "";                            // "0"
String( a );                      // "0"

// как ни странно, даже JSON участвует в обмане
JSON.stringify( a );               // "0"
```

Интересно, что обратные операции (переход от `string` к `number`) не лгут:

```
+"-0";           // -0
Number( "-0" ); // -0
JSON.parse( "-0" ); // -0
```



Поведение метода `JSON.stringify(-0)` с результатом «`0`» выглядит особенно странно, если вы заметите, что оно не соответствует обратным методам: `JSON.parse("-0")` выдает `-0`, как и должно быть.

Кроме того, что преобразование отрицательного нуля к строковому виду только скрывает истинное значение, операторы сравнения тоже *лгут* (намеренно):

```
var a = 0;
var b = 0 / -3;

a == b;      // true
-0 == 0;     // true

a === b;     // true
-0 === 0;    // true

0 > -0;      // false
a > b;        // false
```

Очевидно, если вы хотите отличать `-0` от `0` в своем коде, вы не можете просто полагаться на вывод в консоль разработчика, поэтому придется действовать умнее:

```
function isNegZero(n) {
  n = Number( n );
  return (n === 0) && (1 / n === -Infinity);
}

isNegZero( -0 );      // true
isNegZero( 0 / -3 ); // true
isNegZero( 0 );      // false
```

Для чего же нужен отрицательный нуль, если не считать чистой теории?

В некоторых приложениях разработчики используют абсолютную величину значения для представления одного вида информации (например, скорости движения в кадрах анимации), а знак этого числа – для представления другого вида информации (например, направления этого движения).

Например, если в таком приложении переменная дойдет до нуля и утратит свой знак, вы потеряете информацию о том, в каком направлении происходило движение до обнуления. Сохранение знака нуля предотвращает потенциально нежелательную потерю информации.

Специальное равенство

Как было показано выше, значение `NaN` и значение `-0` проявляют специальное поведение при проверке равенства. Значение `NaN` никогда не равно себе, поэтому приходится использовать `Number.isNaN(..)` (или полифил) из ES6. Кроме того, `-0` тоже вводит в заблуждение и притворяется, что он равен (даже в отношении жесткого равенства `==`, см. главу 4) обычному `0`, так что вам придется использовать ухищрения вроде предложенного выше метода `isNegZero(..)`.

В ES6 появился новый метод, который может использоваться для проверки двух значений на абсолютное равенство без каких-либо исключений. Речь идет о методе `Object.is(..)`:

```
var a = 2 / "foo";
var b = -3 * 0;

Object.is( a, NaN );      // true
Object.is( b, -0 );       // true

Object.is( b, 0 );        // false
```

Приведу довольно простой полифил для `Object.is(..)` в средах до ES6:

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

Вероятно, `Object.is(..)` не стоит использовать в тех случаях, когда `==` или `===` заведомо *безопасны* (см. главу 4), так как операторы наверняка будут работать намного более эффективно, и безусловно, более идиоматичны и распространены. Метод `Object.is(..)` в основном предназначен для этих специальных случаев равенства.

Значения и ссылки

Во многих языках значения могут присваиваться/передаваться по значению или по ссылке в зависимости от используемого синтаксиса. Например, если в C++ вы хотите обновить значение числовой переменной, передаваемой функции, то можете объявить параметр функции вида `int& myNum`. Тогда при передаче переменной (например, `x`) `myNum` будет содержать ссылку на `x`; ссылки работают как специальная разновидность указателей (то есть фактически *синонимы* для других переменных). Если же не объявить ссылочный параметр, то передаваемое значение *всегда* будет копироваться, даже если это сложный объект.

В JavaScript указатели не существуют, а ссылки работают несколько иначе. Одна переменная JS не может хранить ссылку на другую переменную — это попросту невозможно.

Ссылка в JS указывает на (общее) значение, так что, если вы создаете 10 разных ссылок, они всегда будут указывать на одно общее значение; *они никогда не ссылаются/не указывают друг на друга*.

Более того, в JavaScript не существует синтаксических подсказок, управляющих способом передачи (по ссылке или по значению). Вместо этого тип значения управляет только тем, как будет выполняться присваивание – копированием значения или копированием ссылки.

Пример:

```
var a = 2;
var b = a; // `b` всегда содержит копию значения из `a`
b++;
a; // 2
b; // 3

var c = [1,2,3];
var d = c; // `d` – ссылка на общее значение `[1,2,3]`
d.push( 4 );
c; // [1,2,3,4]
d; // [1,2,3,4]
```

Простые значения (то есть скалярные примитивы) *всегда* присваиваются/передаются копированием значения: это `null`, `undefined`, `string`, `number`, `boolean` и `symbol` из ES6.

Составные значения – объекты (включая массивы и все объектные обертки – см. главу 3) и функции – при присваивании или передаче *всегда* создают копию ссылки.

В приведенном примере, поскольку `2` является скалярным примитивом, `a` содержит одну исходную копию значения, а `b` присваивается другая *копия* значения. Но `c` и `d` представляют собой разные ссылки на одно общее значение `[1,2,3]`, которое является составным. Важно заметить, что значение `[1,2,3]` не «принадлежит» ни `c`, ни `d` – это просто равноправные ссылки на значение.

Итак, при использовании любой ссылки для изменения общего массива (`(.push(4))`) изменения распространяются на единственное общее значение, а обе ссылки будут указывать на измененное значение `[1,2,3,4]`.

Так как ссылки указывают на сами значения, а не на переменные, одна ссылка не может использоваться для изменения того, на что ссылается другая ссылка:

```
var a = [1,2,3];
var b = a;
a; // [1,2,3]
b; // [1,2,3]

// позднее
b = [4,5,6];
a; // [1,2,3]
b; // [4,5,6]
```

Выполняя присваивание `b = [4,5,6]`, мы абсолютно ничего не делаем для изменения того, на что сейчас ссылается `a ([1,2,3])`. Чтобы это произошло, переменная `b` должна быть указателем, а не ссылкой на массив, но в JS такой возможности нет.

Чаще всего такие недоразумения происходят с параметрами функций:

```
function foo(x) {
  x.push( 4 );
  x; // [1,2,3,4]

  // позднее
  x = [4,5,6];
  x.push( 7 );
  x; // [4,5,6,7]
}

var a = [1,2,3];

foo( a );
a; // [1,2,3,4], а не [4,5,6,7]
```

При передаче аргумента `a` копия ссылки `a` присваивается `x`. `x` и `a` — разные ссылки, указывающие на одно значение `[1, 2, 3]`. Теперь внутри функции можно использовать эту ссылку для изменения самого значения (`push(4)`). Но когда мы выполняем присваивание `x = [4, 5, 6]`, оно никак не влияет на то, на что указывает исходная ссылка, она все еще указывает на (уже измененный) `[1, 2, 3, 4]`.

Ссылка `x` не может использоваться для изменения того, на что указывает `a`. Можно только изменить содержимое общего значения, на которое указывает как `a`, так и `x`.

Чтобы переменная `a` изменилась и содержала `[4, 5, 6, 7]`, вам не удастся создать новый массив и выполнить присваивание, необходимо изменить существующее значение массива:

```
function foo(x) {  
  x.push( 4 );  
  x; // [1,2,3,4]  
  
  // позднее  
  x.length = 0; // очистить существующий массив на месте  
  x.push( 4, 5, 6, 7 );  
  x; // [4,5,6,7]  
}  
  
var a = [1,2,3];  
  
foo( a );  
  
a; // [4,5,6,7] а не [1,2,3,4]
```

Как видите, `x.length = 0` и `x.push(4, 5, 6, 7)` не создают новый массив, а изменяют существующий общий массив. И конечно, `a` ссылается на новое содержимое `[4, 5, 6, 7]`.

Помните: вы не можете напрямую контролировать/переопределять используемый механизм передачи (копирование значения или копирование ссылки) — эта семантика определяется исключительно типом используемого значения.

Чтобы передать составное значение (например, массив) посредством копирования значения, необходимо создать его копию вручную, чтобы переданная ссылка не продолжала указывать на оригинал. Пример:

```
foo( a.slice() );
```

`slice(..)` без параметров по умолчанию создает совершенно новую (поверхностную) копию массива. Таким образом, передается ссылка только на скопированный массив, а значит, `foo(..)` не сможет повлиять на содержимое `a`.

Чтобы решить обратную задачу – передать скалярное примитивное значение так, чтобы его значение обновлялось как ссылка – необходимо «завернуть» значение в другое составное значение (объект, массив и т. д.), которое *может* быть передано копированием ссылки:

```
function foo(wrapper) {
  wrapper.a = 42;
}

var obj = {
  a: 2
};

foo( obj );
obj.a; // 42
```

Здесь `obj` является оберткой для скалярного примитивного свойства `a`. При передаче `foo(..)` передается копия ссылки `obj`, которая присваивается параметру `wrapper`. После этого ссылка `wrapper` может использоваться для обращения к общему объекту и обновлению его свойства. После завершения функции `obj.a` «увидит» обновленное значение `42`.

И тут возникает другая мысль: если вы хотите передать ссылку на скалярное примитивное значение (например, `2`), нельзя ли упаковать значение в объектную обертку `Number` (см. главу 3)?

Действительно, функции *будет* передана копия ссылки на объект `Number`, но, к сожалению, наличие ссылки на общий объект не даст вам возможности изменить общее примитивное значение, как можно было бы ожидать:

```
function foo(x) {  
    x = x + 1;  
    x; // 3  
}  
  
var a = 2;  
var b = new Number( a ); // эквивалентно `Object(a)`  
  
foo( b );  
console.log( b ); // 2, не 3
```

Проблема в том, что нижележащее скалярное примитивное значение является *неизменяемым* (то же относится к `String` и `Boolean`). Если объект `Number` содержит скалярное примитивное значение 2, этот конкретный объект `Number` не удастся изменить так, чтобы он содержал другое значение; можно только создать совершенно новый объект `Number` с другим значением.

При использовании `x` в выражении `x + 1` нижележащее скалярное примитивное значение 2 распаковывается (извлекается) из объекта `Number` автоматически, так что строка `x = x + 1` совершенно незаметно превращает `x` из общей ссылки на объект `Number` в простое хранилище для скалярного примитивного значения 3 в результате операции сложения `2 + 1`. Следовательно, `b` снаружи продолжает ссылаться на исходный неизмененный/неизменяемый объект `Number`, содержащий значение 2.

К объекту `Number` можно добавлять новые свойства (только не изменяя его внутреннее примитивное значение), так что вы сможете организовать непрямую передачу информации через эти дополнительные свойства.

Впрочем, такие ситуации встречаются довольно редко. Пожалуй, мало кто из разработчиков сочтет такой трюк хорошей практикой программирования.

Вместо того чтобы использовать объектную обертку `Number` подобным образом, лучше использовать «ручное» решение с объектной оберткой (`obj`) из предыдущего фрагмента. Это вовсе не означает, что для таких объектных оберток, как `Number`, нельзя найти творческое применение. Это значит, что в большинстве случаев лучше использовать форму со скалярным примитивным значением.

Ссылки обладают мощными возможностями, но иногда они начинают мешать, а иногда они нужны там, где их нет. Управлять поведением ссылок (выбирать между копированием значений и копированием ссылок) можно только при помощи типа самого значения, так что вам придется косвенно влиять на поведение присваивания/передачи выбором типов используемых значений.

Итоги

В JavaScript массивы представляют собой обычные коллекции значений произвольных типов с числовым индексированием. Строки отчасти являются «массивоподобными», но они обладают другим поведением, и, если вы хотите работать с ними как с массивами, необходима осторожность. К категории чисел в JavaScript принадлежат как «целые» значения, так и значения с плавающей точкой.

Среди примитивных типов определены несколько специальных значений.

Тип `null` имеет всего одно значение `null`; аналогичным образом тип `undefined` имеет единственное значение `undefined`. По сути, `undefined` является значением по умолчанию любой переменной или свойства при отсутствии других значений. Оператор `void` позволяет создать значение `undefined` из любого другого значения.

К числовому типу `number` относятся некоторые специальные значения, такие как `NaN` (вроде бы «не число», но правильнее было бы «недействительное число»), `+Infinity`, `-Infinity` и `-0`.

Простые скалярные примитивы (строки, числа и т. д.) присваиваются/передаются копированием значения, но составные значения (объекты и т. д.) присваиваются/передаются копированием ссылки. Ссылки JavaScript не похожи на ссылки/указатели других языков — они никогда не указывают на другие переменные/ссылки, а только на используемые значения.

3 Встроенные объекты (natives)

В главах 1 и 2 несколько раз упоминались различные встроенные «объекты» — такие, как `String` и `Number`. Рассмотрим их более подробно.

Наиболее часто используемые встроенные вызовы такого рода:

- `String()`
- `Number()`
- `Boolean()`
- `Array()`
- `Object()`
- `Function()`
- `RegExp()`
- `Date()`
- `Error()`
- `Symbol()` — начиная с ES6!

Как видите, эти встроенные «объекты» на самом деле представляют собой встроенные функции.

Если вы пришли в JS из таких языков, как Java, то вызов `String()` в JavaScript покажется вам аналогом конструктора `String(..)`, который вы привыкли использовать для создания строковых значений. Поэтому кратко отметим, что возможны, например, такие операции:

```
var s = new String( "Hello World!" );  
console.log( s.toString() ); // "Hello World!"
```

Действительно, каждая из этих встроенных функций может использоваться как конструктор. Но конструироваться объект может совсем не так, как вы ожидаете:

```
var a = new String( "abc" );  
typeof a; // "object" ... не "String"  
a instanceof String; // true  
Object.prototype.toString.call( a ); // "[object String]"
```

Результатом конструкторной формы создания значения (`new String("abc")`) является объектная обертка для примитивного значения ("abc").

Очень важно, что `typeof` показывает, что эти объекты относятся не к своим конкретным *типам*, а являются подтипами объектного типа.

В дальнейшем для просмотра этой объектной обертки можно воспользоваться командой:

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

Вывод команды зависит от браузера, так как консоли разработчика могут свободно выбирать способ сериализации объекта для просмотра разработчиком.



На момент написания книги вывод в последней версии Chrome выглядел примерно так: `String {0: "a", 1: "b", 2: "c", length: 3, [[PrimitiveValue]]: "abc"}`. Однако старые версии Chrome выводили другой результат: `String {0: "a", 1: "b", 2: "c"}`. Последняя версия Firefox в настоящее время выводит `String ["a", "b", "c"]`, но раньше выводила «`abc`» курсивом; на выводе можно было щелкнуть, чтобы открыть инспектор объектов. Конечно, формат вывода часто изменяется, и в вашем случае результат может выглядеть иначе.

Суть в том, что `new String("abc")` создает объектную обертку для строки `"abc"`, а не само примитивное значение `"abc"`.

Внутреннее свойство `[[Class]]`

Значения с `typeof "object"` (например, массивы) дополнитель но помечаются внутренним свойством `[[Class]]` (считайте это скорее внутренним классификационным признаком, нежели признаком, имеющим отношение к классам из традиционного объектно-ориентированного программирования). К этому свойству нельзя обратиться напрямую, но обычно его можно просмотреть косвенно, вызвав метод по умолчанию `Object.prototype.toString(..)` для этого значения. Пример:

```
Object.prototype.toString.call( [1,2,3] );
// "[object Array]"

Object.prototype.toString.call( /regex-literal/i );
// "[object RegExp]"
```

Итак, для массива из этого примера внутреннее значение `[[Class]]` равно `"Array"`, а для регулярного выражения оно равно `"RegExp"`. В большинстве случаев внутреннее значение `[[Class]]` соответствует встроенному стандартному конструктору (см. ниже), который связан с этим значением. Однако это не всегда так.

Как насчет примитивных значений? Начнем с `null` и `undefined`:

```
Object.prototype.toString.call( null );
// "[object Null]"
```

```
Object.prototype.toString.call( undefined );
// "[object Undefined]"
```

Заметьте, что встроенные конструкторы `Null()` или `Undefined()` не существуют, но тем не менее "`Null`" и "`Undefined`" входят в число внутренних значений `[[Class]]`.

Для других примитивов — таких, как `string`, `number` и `boolean`, — подключается другое поведение, которое обычно называется «упаковкой» (см. «Упаковка»):

```
Object.prototype.toString.call( "abc" );
// "[object String]"
```

```
Object.prototype.toString.call( 42 );
// "[object Number]"
```

```
Object.prototype.toString.call( true );
// "[object Boolean]"
```

В этом фрагменте каждый из простых примитивов автоматически упаковывается соответствующими объектными обертками. Вот почему "`String`", "`Number`" и "`Boolean`" предоставляются как соответствующие внутренние значения `[[Class]]`.



Поведение `toString()` и `[[Class]]` в показанном виде немного изменилось при переходе от ES5 к ES6, но эти подробности будут рассматриваться в книге «`ES6 & Beyond`» этой серии.

Упаковка

Объектные обертки служат очень важной цели. Примитивные значения не имеют свойств или методов, так что для обращения

к `.length` и `.toString()` вам понадобится объектная обертка для значения.

К счастью, JS автоматически *упаковывает* примитивные значения для обслуживания таких обращений:

```
var a = "abc";  
a.length; // 3  
a.toUpperCase(); // "ABC"
```

Итак, если вы собираетесь регулярно обращаться к этим свойствам/методам строковых значений (например, в условии `i < a.length` цикла `for`), может показаться, что разумнее с самого начала использовать объектную форму значения, чтобы движку JS не приходилось неявно создавать ее за вас.

Но, как выясняется, это плохая идея. Браузеры уже давно оптимизировали такие стандартные случаи, как `.length`; это означает, что ваша программа *только замедлится*, если вы попробуете провести «предварительную оптимизацию» с прямым использованием объектной формы (которая не используется на оптимизированном пути).

В общем, нет оснований напрямую использовать объектную форму. Лучше просто положиться на то, что упаковка будет выполняться неявно по мере надобности. Иначе говоря, никогда не используйте конструкции вида `new String("abc")`, `new Number(42)` и т. д. — всегда лучше использовать литералы-примитивы `"abc"` и `42`.

Ловушки при работе с объектными обертками

Прямое использование объектных оберток даже скрывает некоторые ловушки. Если вы когда-либо захотите работать с обертками напрямую, об этих потенциальных проблемах необходимо знать.

Например, возьмем упакованные логические значения:

```
var a = new Boolean( false );  
if (!a) {  
    console.log( "Oops" ); // никогда не выполняется  
}
```

Проблема в том, что вы создаете объектную обертку для значения `false`, но сами объекты являются «истинными» (см. главу 4), так что поведение объекта при использовании прямо противоположно нижележащему значению `false`, конечно, это противоречит нормальным ожиданиям.

Если вы захотите вручную упаковать примитивное значение, используйте функцию `Object(..)` (без ключевого слова `new`):

```
var a = "abc";  
var b = new String( a );  
var c = Object( a );  
  
typeof a; // "string"  
typeof b; // "object"  
typeof c; // "object"  
  
b instanceof String; // true  
c instanceof String; // true  
  
Object.prototype.toString.call( b ); // "[object String]"  
Object.prototype.toString.call( c ); // "[object String]"
```

И снова напрямую использовать упакованные объектные обертки (как `b` и `c` в этом примере) обычно не рекомендуется, но иногда попадаются редкие ситуации, в которых они могут оказаться полезными.

Распаковка

Если у вас имеется объектная обертка и вы хотите получить нижележащее примитивное значение, используйте метод `valueOf()`:

```
var a = new String( "abc" );
var b = new Number( 42 );
var c = new Boolean( true );

a.valueOf(); // "abc"
b.valueOf(); // 42
c.valueOf(); // true
```

Распаковка также может выполняться неявно, когда объектная обертка используется в контексте, требующем примитивного значения. Этот процесс (преобразование) будет подробно рассматриваться в главе 4, но если кратко:

```
var a = new String( "abc" );
var b = a + ""; // `b` содержит распакованное примитивное
                // значение "abc"

typeof a;        // "object"
typeof b;        // "string"
```

Встроенные объекты как конструкторы

Для значений — массивов, объектов, функций и регулярных выражений почти всегда рекомендуется использовать литеральную форму для создания значений, но литеральная форма создает такой же объект, как форма конструктора (то есть неупакованного значения вообще нет). Как и для других встроенных объектов, в общем случае использовать формы конструкторов нежелательно, если только вы не уверены, что они действительно необходимы, прежде всего из-за исключений и потенциальных ловушек. Поверьте, вам *не захочется* иметь с ними дела.

Array(..)

```
var a = new Array( 1, 2, 3 );
a; // [1, 2, 3]
```

```
var b = [1, 2, 3];
b; // [1, 2, 3]
```



Перед конструктором `Array()` не обязательно ставить ключевое слово `new`. Даже если опустить `new`, конструктор ведет себя так, словно оно присутствует. Таким образом, `Array(1, 2, 3)` приведет к тому же результату, что и `new Array(1, 2, 3)`.

У конструктора `Array` существует специальная форма, в которой при передаче только одного аргумента это значение интерпретируется как длина для «предварительного выделения памяти под массив» (в своем роде).

Это просто ужасная идея. Во-первых, вы можете случайно споткнуться об эту форму, потому что о ней легко забыть.

Но что еще важнее, никакого предварительного выделения памяти под массив нет и в помине. Вместо этого создается пустой массив, свойству `length` которого присваивается заданное вами числовое значение.

Массив, который не содержит явных значений в своих элементах, но обладает свойством `length`, подразумевающим существование таких элементов, является необычной и экзотической структурой данных в JS, у которой очень странное и запутанное поведение. Возможность создания таких значений происходит исключительно от устаревшей, отмершей функциональности («массиво-подобные объекты» как объекты `arguments`).



Массивы, содержащие как минимум один «пустой элемент», часто называются «разреженными» (sparse).

Проблема усугубляется тем, что здесь консоли разработчика снова по-разному представляют такие объекты; это создает еще больше путаницы.

Пример:

```
var a = new Array( 3 );
a.length; // 3
a;
```

В Chrome (на момент написания книги) значение `a` выводится в виде `[undefined x 3]`. Это в высшей степени неудачный выбор — он утверждает, что в элементах массива содержатся три значения `undefined`, тогда как в действительности эти элементы не существуют!

Чтобы наглядно представить разницу, попробуйте выполнить следующий пример:

```
var a = new Array( 3 );
var b = [ undefined, undefined, undefined ];
var c = [];
c.length = 3;

a;
b;
c;
```



Как показывает этот пример для переменной `c`, пустые элементы в массиве могут появляться и после создания массива. При изменении длины массива и выходе за пределы количества фактически определенных значений элементов вы неявно создаете пустые элементы. Даже при вызове `delete b[1]` в этом фрагменте в середине `b` появится пустой элемент.

Для `b` (в настоящее время в Chrome) выводится сериализация `[undefined, undefined, undefined]` в отличие от `[undefined x 3]` для `a` и `c`. Запутались? Да, как и все остальные.

Что еще хуже, на момент написания книги Firefox выводит `[, ,]` для `a` и `c`. Заметили, почему это особенно странно? Присмотриесь внимательнее. Три запятые подразумевают четыре элемента, а не три элемента, как можно было бы ожидать.

Что-что?.. Firefox помещает дополнительную запятую , в конец сериализации, потому что в ES5 завершающие запятые в списках (значения массивов, списки свойств и т. д.) разрешены (они отбрасываются и игнорируются). Таким образом, если введете в своей программе или в консоли значение [, , ,], то получите базовое значение вида [, ,] (то есть массив с тремя пустыми элементами). В пользу этого решения, хотя оно и выглядит странно при чтении консоли разработчика, говорит то, что оно обеспечивает правильную работу копирования/вставки.

Качаете головой или закатываете глаза? Вы не одиноки! Чудно все это.



Похоже, в новых версиях Firefox вывод в этой ситуации меняется на Array [<3 empty slots>], что безусловно намного лучше, чем [, ,].

К сожалению, все еще хуже. а и b в приведенном фрагменте не только запутывают консольный вывод, но и ведут себя одинаково в одних случаях, но по-разному в других:

```
a.join( "-"); // "--"  
b.join( "-"); // "--"  
  
a.map(function(v,i){ return i; }); // [ undefined x 3 ]  
b.map(function(v,i){ return i; }); // [ 0, 1, 2 ]
```

Вызов a.map() завершается *неудачей*, потому что элементы на самом деле не существуют, так что map(..) нечего перебирать. Метод join(..) работает иначе. По сути, можно считать, что он реализован примерно так:

```
function fakeJoin(arr,connector) {  
    var str = "";  
    for (var i = 0; i < arr.length; i++) {  
        if (i > 0) {  
            str += connector;
```

```
        }
        if (arr[i] !== undefined) {
            str += arr[i];
        }
    }
    return str;
}

var a = new Array( 3 );
fakeJoin( a, "-" ); // "--"
```

Как видите, `join(..)` просто *считает*, что элементы существуют, и выполняет перебор до длины `length`. Как бы ни была устроена внутренняя реализация `map(..)`, она (очевидно) не делает таких предположений, так что результат странного «массива с пустыми элементами» оказывается неожиданным и, скорее всего, приведет к сбою.

Итак, если вы хотите создать массив со значениями `undefined` (а не «пустыми элементами»), как это сделать (кроме как вручную)?

```
var a = Array.apply( null, { length: 3 } );
a; // [ undefined, undefined, undefined ]
```

Запутались? Еще бы. Сейчас расскажу, как это примерно работает.

Метод `apply(..)` может использоваться со всеми функциями. Он вызывает функцию, с которой используется, но особым образом.

Первый аргумент содержит объектную привязку `this`. В данном случае нам это не нужно, поэтому аргументу присваивается `null`. Второй аргумент должен содержать массив (или что-то *похожее*, то есть объектный «аналог массива»). Содержимое этого массива «развертывается» в набор аргументов соответствующей функции.

Итак, `Array.apply(..)` вызывает функцию `Array(..)` и развертывает значения (из `{ length: 3 }`) в ее аргументах.

Внутри `apply(..)` можно представить себе еще один цикл `for` (как в примере `join(..)` из приведенного примера), который идет от `0` до `length` (3 в нашем случае), не включая последнего.

Для каждого индекса из объекта читается соответствующий ключ. Таким образом, если внутри функции `apply` аналогу массива будет присвоено имя `arr`, обращения будут иметь вид `arr[0]`, `arr[1]` и `arr[2]`. Конечно, ни одно из этих свойств не существует в значении объекта `{ length: 3 }`, так что при обращениях ко всем трем свойствам будет возвращено значение `undefined`.

Иначе говоря, в итоге она вызывает `Array(..)` примерно в таком виде: `Array(undefined, undefined, undefined)`. Так мы получили массив, заполненный значениями `undefined`, а не только (очень странные) пустые элементы.

Хотя `Array.apply(null, { length: 3 })` кажется непонятным и слишком длинным способом создания массивов, заполненных значениями `undefined`, эта запись *неизмеримо* лучше и надежнее того, что вы получите при использовании коварного `Array(3)` с его пустыми элементами.

Мораль: *никогда, ни при каких обстоятельствах* не пытайтесь намеренно создавать и использовать массивы с пустыми элементами. Просто не делайте этого. Они совершенно безумны.

Object(..), Function(..) и RegExp(..)

Конструкторы `Object(..)`/`Function(..)`/`RegExp(..)` тоже обычно не обязательны (а следовательно, их лучше избегать, если только ситуация не требует обратного):

```
var c = new Object();
c.foo = "bar";
c; // { foo: "bar" }

var d = { foo: "bar" };
```

```
d; // { foo: "bar" }

var e = new Function( "a", "return a * 2;" );
var f = function(a) { return a * 2; }
function g(a) { return a * 2; }

var h = new RegExp( "^a*b+", "g" );
var i = /^a*b+/g;
```

Практически нет никаких причин когда-либо использовать форму конструктора `new Object()`, особенно потому, что она заставляет вас добавлять свойства одно за другим (вместо одновременного добавления сразу многих, как в форме объектного литерала).

Конструктор `Function` полезен только в редчайших случаях, когда параметры и/или тело функции должны определяться динамически. Не стоит относиться к `Function(..)` как к альтернативной форме `eval(..)`. Вам практически никогда не придется динамически определять функции подобным образом.

Регулярные выражения настоятельно рекомендуется определять в форме литерала `(/^a*b+/g)` — не только ради простоты синтаксиса, но и по соображениям быстродействия; движок JS заранее компилирует и кэширует их перед выполнением кода. Но в отличие от других форм конструкторов, которые встречались вам до настоящего момента, у `RegExp(..)` есть разумное практическое применение: динамическое определение шаблона для регулярного выражения:

```
var name = "Kyle";
var namePattern = new RegExp( "\\b(?: " + name + ")+\\b", "ig" );

var matches = someText.match( namePattern );
```

Подобные ситуации время от времени встречаются в программах JS. Используйте форму `new RegExp("pattern","flags")`.

Date(..) и Error(..)

Конструкторы `Date(..)` и `Error(..)` намного полезнее других встроенных объектов, потому что у них нет формы литерала.

Для создания объекта даты необходимо использовать `new Date()`. Конструктор `Date(..)` получает необязательные аргументы, определяющие используемые дату/время; если они не указаны, используются текущие дата/время.

Бессспорно, конструирование объекта даты чаще всего применяется для получения текущего значения временного штампа UNIX (целого количества секунд, прошедших с 1 января 1970 года). Это можно сделать вызовом `getTime()` для экземпляра объекта даты.

Но самый простой способ основан на вызове статической вспомогательной функции, определенной в ES5: `Date.now()`. Создать полифил для версий до ES5 несложно:

```
if (!Date.now) {
  Date.now = function(){
    return (new Date()).getTime();
  };
}
```



Если вызвать `Date()` без `new`, вы получите строковое представление текущих даты/времени. Точная форма этого представления не задана в спецификации языка, хотя браузеры обычно выдают строку вида «`Fri Jul 18 2014 00:31:02 GMT-0500 (CDT)`».

Конструктор `Error(..)` (как и `Array()` — см. выше) ведет себя одинаково как с ключевым словом `new`, так и без него.

Главная причина, по которой вы станете создавать объекты ошибок в своей программе, — это сохранение текущего контекста

стека выполнения в объекте (в большинстве сред JS после конструирования предоставляется в виде свойства `.stack`, доступного только для чтения).

Контекст стека включает стек вызовов функций и номер строки, в которой был создан объект ошибки. Эта информация заметно упрощает отладку.

Чаще всего такие объекты ошибок используются с оператором `throw`:

```
function foo(x) {  
  if (!x) {  
    throw new Error( "x wasn't provided" );  
  }  
  // ..  
}
```

Экземпляры объектов ошибок обычно содержат как минимум свойство `message`, а иногда и другие свойства (которые должны рассматриваться как доступные только для чтения), такие как `type`. Тем не менее, если не считать анализа упоминавшегося выше свойства `stack`, обычно лучше просто вызвать `toString()` для объекта ошибки (явно или косвенно посредством преобразования типа, см. главу 4) для получения удобно отформатированного сообщения об ошибке.



Кроме того, помимо общей обертки `Error(..)`, существует несколько оберток для конкретных разновидностей ошибок: `Evaluator(..)`, `RangeError(..)`, `ReferenceError(..)`, `SyntaxError(..)`, `TypeError(..)` и `URIError(..)`. Тем не менее использовать эти конкретные ошибки вручную вам практически никогда не придется. Они используются автоматически в том случае, если в вашей программе действительно произошло реальное исключение (например, при обращении к необъявленной переменной произошла ошибка `ReferenceError`).

Symbol(..)

В ES6 появился новый примитивный тип значений `Symbol`. Значения этого типа — *символические имена* — представляют собой специальные «уникальные» (без гарантий полной уникальности!) значения, которые могут использоваться как свойства объектов с минимальным риском конфликтов. Они в основном создавались для специальных встроенных аспектов поведения конструкций ES6, но вы также можете определять собственные символические имена.

Символические имена могут использоваться как имена свойств, но вы не сможете просмотреть фактическое значение символического имени или обратиться к нему ни из своей программы, ни с консоли разработчика. Если попытаться вычислить символическое имя на консоли разработчика, будет выведен результат вида `Symbol(Symbol.create)`.

В ES6 есть несколько заранее определенных символьических имен, для обращения к которым используются статические свойства объекта функции `Symbol`: `Symbol.create`, `Symbol.iterator` и т. д. Пример их использования:

```
obj[Symbol.iterator] = function(){ /*..*/ };
```

Чтобы определить собственные пользовательские символьические имена, используйте встроенный объект `Symbol(..)`. Встроенный «конструктор» `Symbol(..)` унаследован тем, что с ним нельзя использовать ключевое слово `new`, если вы попытаетесь это сделать, произойдет ошибка:

```
var mysym = Symbol( "my own symbol" );
mysym;           // Symbol(my own symbol)
mysym.toString(); // "Symbol(my own symbol)"
typeof mysym;    // "symbol"

var a = { };
a[mysym] = "foobar";
```

```
Object.getOwnPropertySymbols( a );
// [ Symbol(my own symbol) ]
```

Хотя символические имена не являются приватными (`Object.getOwnPropertySymbols(..)` получает информацию об объекте и предоставляет символические имена вполне открыто), в первую очередь, они должны применяться для приватных или специальных свойств. Для большинства разработчиков они могут занять место имен свойств с префиксами `_` (подчеркивание), что по соглашениям почти всегда означает: «Внимание! Приватное/специальное/внутреннее свойство, не трогать!».



Символические имена *не* являются объектами, это простые скалярные примитивы.

Встроенные прототипы

Каждый из встроенных конструкторов имеет собственный объект `.prototype` – `Array.prototype`, `String.prototype` и т. д. Эти объекты обладают специальным поведением, уникальным для их конкретного подтипа объекта.

Например, для всех строковых объектов и, если брать шире, строковых примитивов (через упаковку) доступно поведение по умолчанию в виде методов, определенных для объекта `String.prototype`.



По документационным соглашениям `String.prototype.XYZ` сокращается до `String#XYZ`; это относится ко всем прототипам.

- `String#indexOf(..)` – находит позицию вхождения подстроки в другой строке.

- `String#charAt(..)` – обращается к символу в позиции строки.
- `String#substr(..), String#substring(..) и String#slice(..)` – извлекает часть существующей строки с созданием новой строки.
- `String#toUpperCase() и String#toLowerCase()` – создает новую строку, которая преобразуется к верхнему или нижнему регистру.
- `String#trim()` – создает новую строку, из которой удалены все начальные или конечные пропуски.

Ни один из методов не изменяет строку «на месте». Все изменения (такие, как преобразования регистра или удаление начальных/конечных пропусков) создают новое значение на базе существующего.

Благодаря делегированию прототипов эти методы доступны для любых строковых значений:

```
var a = " abc ";
a.indexOf( "c" ); // 3
a.toUpperCase(); // " ABC "
a.trim(); // "abc"
```

Другие прототипы конструкторов содержат поведение, относящееся к их типам, как, например, `Number#toFixed(..)` (преобразование числа в строку с фиксированным количеством цифр) и `Array#concat(..)` (слияние массивов). Всем функциям доступны `apply(..)`, `call(..)` и `bind(..)`, потому что они определены в прототипе `Function.prototype`.

Однако некоторые встроенные прототипы не являются *простыми* объектами:

```
typeof Function.prototype; // "function"
Function.prototype(); // пустая функция!
```

```
RegExp.prototype.toString();          // "/(?:)/" -- пустое
                                         // регулярное выражение
"abc".match( RegExp.prototype );    // [""]
```

Эти прототипы даже можно изменять (не просто добавлять свойства, к чему вы, возможно, уже привыкли), и это особенно плохая мысль:

```
Array.isArray( Array.prototype );    // true
Array.prototype.push( 1, 2, 3 );      // 3
Array.prototype;                    // [1,2,3]

// не оставляйте его в таком виде, иначе начнутся странности!
// верните `Array.prototype` к пустому значению
Array.prototype.length = 0;
```

Как видите, `Function.prototype` — функция, `RegExp.prototype` — регулярное выражение, а `Array.prototype` — массив. Круто, да?

Прототипы как значения по умолчанию

Тот факт, что `Function.prototype` является пустой функцией, `RegExp.prototype` — «пустым» регулярным выражением (например, без совпадения), а `Array.prototype` — пустым массивом, делает их удобными «значениями по умолчанию», которые могут быть присвоены переменным, если этим переменным еще не присвоено значение подходящего типа.

Пример:

```
function isThisCool(vals,fn,rx) {
  vals = vals || Array.prototype;
  fn = fn || Function.prototype;
  rx = rx || RegExp.prototype;

  return rx.test(
    vals.map( fn ).join( "" )
  );
}
```

```
isThisCool();           // true

isThisCool(
  ["a","b","c"],
  function(v){ return v.toUpperCase(); },
  /D/
);                   // false
```



Начиная с ES6 больше не нужно использовать синтаксический трюк `vals = vals || ..` для значений по умолчанию (см. главу 4), потому что значения по умолчанию для параметров могут задаваться при помощи встроенного синтаксиса в объявлении функции (см. главу 5).

Одно из дополнительных преимуществ такого подхода заключается в том, что `.prototype` уже созданы и встроены в язык; таким образом, они создаются *только единожды*. С другой стороны, при использовании `[]`, `function(){}()` и `/(?:)/` для этих значений по умолчанию приведет (скорее всего, но это зависит от реализаций движка) к повторному созданию этих значений при *каждом вызове* `isThisCool(..)` — и их уничтожению в ходе уборки мусора. Это приведет к неэффективному расходованию памяти/процессора.

Также будьте внимательны и не используйте `Array.prototype` в качестве значения по умолчанию, которое должно изменяться в дальнейшем. В этом примере `vals` используется только для чтения, но при попытке внести изменения в `vals` «на месте» вы будете изменять `Array.prototype`. Это создаст проблемы, о которых говорилось ранее!



Хотя мы указали, что встроенные прототипы могут принести практическую пользу, будьте внимательны, особенно при попытках как-либо изменять их. За дополнительной информацией обращайтесь к разделу «Встроенные прототипы» в приложении А.

Итоги

JavaScript предоставляет объектные обертки для примитивных значений (`String`, `Number`, `Boolean` и т. д.). Объектные обертки предоставляют значениям доступ к поведению, соответствующему каждому подтипу объекта (`String#trim()` и `Array#concat(..)`).

Если у вас имеется простое скалярное примитивное значение (например, `"abc"`) и вы обращаетесь к его свойству `length` или вызываете некий метод `String.prototype`, JS автоматически «упаковывает» значение (заключает его в соответствующую объектную обертку), для того чтобы реализовать обращение к свойству/методу.

4

Преобразование типов

Теперь, когда вы лучше понимаете типы и значения JavaScript, обратимся к весьма неоднозначной теме: преобразованию типов. Как упоминалось в главе 1, споры относительно того, является ли преобразование типов полезной возможностью или недостатком языка (или чем-то средним), идут с первых дней. Если вы читали другие популярные книги по JS, то господствующее мнение вам уже известно: преобразование типов — это мистический, злокозненный, невразумительный механизм... да и вообще это откровенно плохая мысль.

Мы не будем здесь убегать от преобразования типов (потому что так делают все остальные), а попробуем приблизиться к тому, чего вы не понимаете, и постараться разобраться в проблеме. Наша задача — полностью исследовать все плюсы и минусы (да, плюсы тоже есть!) системы преобразования типов, чтобы вы могли принять обоснованное решение относительно его уместности в вашей программе.

Преобразование значений

Преобразование значения от одного типа к другому часто делится на явное (выполняемое напрямую) и неявное (обусловленное правилами использования этого значения).



Может, это и не очевидно, но преобразования типов JavaScript всегда дают одно из скалярных примитивных значений (см. главу 2), такое как строка, число или логическое значение. Не существует преобразования, которое бы давало сложное значение (например, объект или функцию). В главе 3 рассматривается «упаковка» — заключение примитивных значений в их объектные аналоги, но процесс упаковки не является преобразованием типа в точном смысле.

Также эти термины часто различаются следующим образом: явное преобразование типов выполняется в языках со статической типизацией на стадии компиляции, а неявное преобразование типов выполняется на стадии выполнения в языках с динамической типизацией. Впрочем, в JavaScript большинство разработчиков использует общий термин «преобразование типов».

Различия между ними должны быть очевидны: «явное преобразование типа» происходит тогда, когда из кода становится ясно, что тип преобразуется намеренно, тогда как «неявное преобразование» выполняется тогда, когда оно является менее очевидным побочным эффектом другой явной операции.

Например, возьмем следующие два примера преобразования типов:

```
var a = 42;  
var b = a + "";           // неявное преобразование  
var c = String( a );    // явное преобразование
```

Для `b` преобразование типа выполняется неявно, потому что у оператора `+` один из операндов является строковым значением `("")`. В такой ситуации оператор выполняет операцию конкatenации строк (то есть слияние двух строк), у которой имеется (*скрытый*) *побочный эффект*: он заставляет преобразовать значение `42` в его строковый эквивалент: `"42"`.

С другой стороны, функция `String(..)` наглядно показывает, что значение `a` явно преобразуется в строковое представление.

Оба способа достигают одного эффекта: `42` превращается в `"42"`. Но именно этот момент находится в центре ожесточенных споров по поводу преобразования типов JavaScript.



Технически помимо стилистических различий существуют некоторые нетривиальные нюансы в поведении. Они будут более подробно рассмотрены позднее в этой главе, в разделе «Неявные преобразования: `String <--> Number`».

Термины «явный» и «неявный», «очевидный» и «скрытый побочный эффект» *относительны*.

Если вы точно знаете, что делает `+ ""`, и намеренно используете этот факт для преобразования к строке, операция может показаться достаточно «явной». И наоборот, если вы никогда не видели, как функция `String(..)` используется для преобразования строк, ее поведение может показаться достаточно загадочным, чтобы этот эффект показался «неявным».

Но мы обсуждаем «явность» и «неявность» на основании наиболее вероятных мнений среднего более или менее информированного разработчика, который *не является экспертом или знаком спецификации JS*. В какой бы степени вы ни относились к этой категории, стоит рассматривать приводимые здесь рассуждения именно под этим углом зрения.

Но помните: редко когда вы становитесь единственным читателем написанного вами кода. Даже если вы досконально разбираетесь во всех тонкостях JS, подумайте, что почувствует менее информированный коллега, когда будет читать ваш код. Будет ли «явность» или «неявность» для него означать то же самое, что и для вас?

Абстрактные операции

Прежде чем разбираться в явных и неявных преобразованиях, необходимо изучить базовые правила, управляющие тем, как значения превращаются в строки, числа или логические значения. Спецификация ES5 в разделе 9 определяет несколько «абстрактных операций» (хитроумный термин, обозначающий «операции только для внутреннего использования») с правилами преобразования значений. Особое внимание будет уделено `ToString`, `ToNumber` и `ToBoolean`, и в меньшей степени `ToPrimitive`.

ToString

Когда любое нестроковое значение преобразуется в строковое представление, преобразование выполняется абстрактной операцией `ToString` из раздела 9.8 спецификации.

Встроенные примитивные значения содержат естественное преобразование к строковому виду: `null` превращается в `"null"`, `undefined` превращается в `"undefined"`, `true` превращается в `"true"`. Числа обычно выражаются в естественном виде, привычном нам, но как упоминалось в главе 2, очень малые или очень большие числа представляются в экспоненциальной форме:

```
// `1.07` умножается на `1000` семь раз
var a = 1.07 * 1000 * 1000 * 1000 * 1000 * 1000 * 1000;

// семь раз по три цифры = 21 цифра
a.toString(); // "1.07e21"
```

Для обычных объектов, если только вы не укажете собственную реализацию, реализация `toString()` по умолчанию (`Object.prototype.toString()`) возвращает внутренний `[[Class]]` (см. главу 3), например `"[object Object]"`.

Но как было показано ранее, если объект содержит собственный метод `toString()` и этот объект используется в строковом кон-

тексте, автоматически будет вызвана его реализация `toString()` и будет использован строковый результат этого вызова.



Способ преобразования объекта в строку формально проходит через абстрактную операцию `ToPrimitive` (спецификация ES5, раздел 9.1), но эти тонкости рассматриваются более подробно в разделе «`ToNumber`» далее в этой главе, поэтому здесь мы их пропускаем.

Массивы имеют переопределенную версию по умолчанию `toString()`, которая выполняет (строковую) конкатенацию всех своих значений (каждое из которых также преобразуется к строковой форме), разделяя их ", ":"

```
var a = [1,2,3];
a.toString(); // "1,2,3"
```

И снова метод `toString()` может вызываться явно, или же он будет вызван автоматически при использовании нестрокового значения в строковом контексте.

Преобразование JSON в строку

Другая задача, также тесно связанная с `ToString`, — использование метода `JSON.stringify(..)` для сериализации значения в JSON-совместимое строковое значение.

Важно заметить, что задача `stringify(..)` несколько отличается от преобразования в строку. Но поскольку она связана с правилами `ToString`, упомянутыми выше, мы ненадолго отвлечемся на рассмотрение преобразования JSON.

Для большинства простых значений преобразование JSON ведет себя практически так же, как преобразования `toString()`, если не считать того, что результатом сериализации всегда является строка:

```
JSON.stringify( 42 ); // "42"  
JSON.stringify( "42" ); // ""42"" (строка с заключенным  
// в кавычки строковым значением)  
JSON.stringify( null ); // "null"  
JSON.stringify( true ); // "true"
```

Любое JSON-безопасное значение может быть преобразовано вызовом `JSON.stringify(..)`. Но что считать «JSON-безопасным»? Любое значение, которое имеет действительное представление в формате JSON.

Возможно, будет проще рассматривать значения, которые *не* являются JSON-безопасными. Несколько примеров: `undefined`, функции, символические имена (ES6+) и объекты с циклическими ссылками (свойства в объектной структуре ссылаются друг на друга, создавая бесконечный цикл). Все эти значения недопустимы в стандартной структуре JSON, прежде всего потому, что они не портируются на другие языки, потребляющие значения JSON.

Метод `JSON.stringify(..)` автоматически опускает обнаруженные значения `undefined`, `function` и `symbol`. Если такое значение будет найдено в массиве, оно заменяется `null` (чтобы положение информации в массиве не изменилось). Если значение обнаруживается в свойстве объекта, это свойство просто исключается.

Пример:

```
JSON.stringify( undefined ); // undefined  
JSON.stringify( function(){ } ); // undefined  
  
JSON.stringify(  
  [1,undefined,function(){},4]  
); // "[1,null,null,4]"  
JSON.stringify(  
  { a:2, b:function(){ } }  
); // {"a":2}
```

Но если вы попытаетесь вызвать `JSON.stringify(..)` с объектом, содержащим циклические ссылки, произойдет ошибка.

У преобразования JSON есть особое поведение: если у значения `object` определен метод `toJSON()`, сначала будет вызван этот метод. Он дает значение, которое должно использоваться для сериализации.

Если вы собираетесь применить преобразование JSON к объекту, который может содержать недействительные значения JSON, или если объект содержит значения, недопустимые для сериализации, определите метод `toJSON()`, который возвращает JSON-безопасную версию объекта.

Пример:

```
var o = { };

var a = {
  b: 42,
  c: o,
  d: function(){}
};

// создать циклическую ссылку в `a`
o.e = a;

// выдаст ошибку из-за циклической ссылки
// JSON.stringify( a );

// определить нестандартную сериализацию значения JSON
a.toJSON = function() {
  // включить для сериализации только свойство `b`
  return { b: this.b };
};

JSON.stringify( a ); // {"b":42}
```

Существует очень распространенное заблуждение, будто метод `toJSON()` должен возвращать результат преобразования JSON. Вероятно, оно ошибочно, если только вы не хотите действительно преобразовать к строковой форме саму строку (обычно нет!). Метод `toJSON()` должен возвращать обычное значение (любого

типа), соответствующее конкретной ситуации, а метод `JSON.stringify(..)` обеспечит его преобразование в строку.

Другими словами, метод `toJSON()` следует интерпретировать как «приведение к JSON-безопасному значению, подходящему для преобразования в строку», а не «преобразование в строку JSON», как ошибочно полагают многие разработчики.

Возьмем следующий фрагмент:

```
var a = {
  val: [1,2,3],
  // Наверное, правильно.
  toJSON: function(){
    return this.val.slice( 1 );
  }
};

var b = {
  val: [1,2,3],
  // Наверное, ошибка.
  toJSON: function(){
    return "[" +
      this.val.slice( 1 ).join() +
    "]";
  }
};

JSON.stringify( a ); // "[2,3]"
JSON.stringify( b ); // ""[2,3]""
```

Во втором вызове в строку преобразуется возвращенная строка, а не сам массив — вероятно, это не то, что вы хотели.

Раз уж речь зашла о `JSON.stringify(..)`, обсудим некоторые малоизвестные возможности, которые могут оказаться очень полезными.

В `JSON.stringify(..)` может передаваться необязательный второй аргумент, который называется *заменителем* (replacer). Этим

аргументом может быть массив или функция. Он предназначен для настройки рекурсивной сериализации объекта и предоставляет механизм фильтрации свойств, которые должны (или не должны) включаться в результат, по аналогии с тем, как `toJSON()` готовит значение для сериализации.

Если заменитель является массивом, это должен быть массив строк, каждая из которых задает имя свойства, разрешенного для включения в сериализацию объекта. Если какое-то свойство не входит в этот список, оно пропускается.

Если заменитель является функцией, эта функция сначала будет вызвана для самого объекта, а затем для каждого свойства в объекте; каждый раз ей передаются два аргумента, *ключ* и *значение*. Чтобы пропустить ключ в сериализации, верните `undefined`. В противном случае верните значение.

```
var a = {  
  b: 42,  
  c: "42",  
  d: [1,2,3]  
};  
  
JSON.stringify( a, ["b","c"] ); // "{"b":42,"c":"42"}"  
  
JSON.stringify( a, function(k,v){  
  if (k !== "c") return v;  
} );  
// {"b":42,"d":[1,2,3]}"
```



В случае заменителя-функции аргумент ключа *k* не определен для первого вызова (при котором передается сам объект *a*). Команда `if` отфильтровывает свойство с именем «*c*». Преобразование в строку выполняется рекурсивно, так что каждое из значений массива `[1,2,3]`, то есть (1, 2 и 3), передается заменителю в аргументе *v*, а индексы (0, 1 и 2) передаются в аргументе *k*.

`JSON.stringify(..)` также может передаваться третий необязательный аргумент *отступ*, который используется для украшения

вывода. Отступ может быть положительным целым числом — тогда он указывает, сколько пробелов должно использоваться на каждом уровне отступов. Также отступ может быть строкой; в этом случае на каждом уровне отступов используются его начальные символы (до 10):

```
var a = {  
  b: 42,  
  c: "42",  
  d: [1,2,3]  
};  
  
JSON.stringify( a, null, 3 );  
// "{  
//   "b": 42,  
//   "c": "42",  
//   "d": [  
//     1,  
//     2,  
//     3  
//   ]  
// }"  
  
JSON.stringify( a, null, "-----" );  
// "{  
// -----"b": 42,  
// -----"c": "42",  
// -----"d": [  
// -----1,  
// -----2,  
// -----3  
// -----]  
// }"
```

Еще раз напомню, что метод `JSON.stringify(..)` не является прямой формой преобразования типа. Здесь мы рассмотрели его по двум причинам, связывающим его поведение с преобразованием `ToString`:

1. Значения `string`, `number`, `boolean` и `null` преобразуются для JSON практически так же, как они преобразуются в строковые значения по правилам абстрактной операции `ToString`.

- Если передать `JSON.stringify(..)` объектное значение, и у этого объекта определен метод `toJSON()`, он автоматически будет вызываться для «приведения» значения в JSON-безопасное перед преобразованием в строковую форму.

ToNumber

Если любое нечисловое значение используется в контексте, в котором оно должно быть числом (например, в математической операции), спецификация ES5 определяет абстрактную операцию `ToNumber` в разделе 9.3.

Например, `true` преобразуется в `1`, а `false` преобразуется в `0`, `undefined` превращается в `NaN`, но (как ни странно) `null` превращается в `0`.

`ToNumber` для строкового значения большей частью работает по аналогии с правилами/синтаксисом числовых литералов (см. главу 3). Если попытка завершается неудачей, то результат равен `NaN` (вместо синтаксической ошибки, как в случае с числовыми литералами). Одно из отличий заключается в том, что восьмеричные числа с префиксом `0` в этой операции обрабатываются не как восьмеричные, а как обычные десятичные, хотя такие восьмеричные числа являются действительными, как числовые литералы (см. главу 2).



Различия между грамматикой числовых литералов и `ToNumber` для строкового значения достаточно тонкие и неочевидные, поэтому здесь они рассматриваться не будут. За дополнительной информацией обращайтесь к разделу 9.3.1 спецификации ES5.

Объекты (и массивы) сначала преобразуются в свои эквиваленты среди примитивных значений, а полученное значение (если

это примитив, но еще не число) преобразуется в число по только что упомянутым правилам `ToNumber`.

Чтобы выполнить преобразование к эквивалентному примитивному значению, абстрактная операция `ToPrimitive` (спецификация ES5, раздел 9.1) проверяет соответствующее значение (с использованием внутренней операции `DefaultValue` — спецификация ES5, раздел 8.12.8) на наличие метода `valueOf()`. Если метод `valueOf()` доступен и возвращает примитивное значение, это значение используется для преобразования типа. Если нет, то `toString()` предоставляет значение для преобразования (если возможно).

Если ни одна операция не может предоставить примитивное значение, выдается ошибка `TypeError`.

В ES5 можно создать такой «непреобразуемый» объект (без `valueOf()` и `toString()`), если его `[[Prototype]]` содержит значение `null`, обычно создаваемый вызовом `Object.create(null)`.



Позднее в этой главе мы подробно рассмотрим процесс преобразования в числа, но для следующего фрагмента кода просто считайте, что функция `Number(..)` решает эту задачу.

Пример:

```
var a = {
  valueOf: function(){
    return "42";
  }
};

var b = {
  toString: function(){
    return "42";
  }
};
```

```
var c = [4,2];
c.toString = function(){
    return this.join( "" ); // "42"
};

Number( a );                // 42
Number( b );                // 42
Number( c );                // 42
Number( "" );                // 0
Number( [] );                // 0
Number( [ "abc" ] );        // NaN
```

ToBoolean

Теперь немного поговорим о том, как в JS себя ведут логические значения. Вокруг этой темы много путаницы и заблуждений, поэтому будьте внимательны!

Прежде всего, в JS существуют ключевые слова `true` и `false`, и они ведут себя именно так, как можно ожидать от логических значений. Существует распространенное заблуждение, будто значения `1` и `0` идентичны `true/false`. Возможно, в других языках это действительно так, но в JS числа — это числа, а логические значения — это логические значения. `1` можно преобразовать в `true` (и наоборот), а `0` в `false` (и наоборот). Однако это не одно и то же.

Ложные значения

Но это еще не все. Необходимо разобраться в том, как ведут себя другие значения, кроме этих двух, при преобразовании в их логический эквивалент.

Все значения JavaScript можно разделить на две категории:

1. Значения, которые при преобразовании в `boolean` дают `false`.
2. Все остальные (которые, очевидно, дают `true`).

Я вовсе не пытаюсь острить. В спецификации JS определен конкретный узкий набор значений, которые при преобразовании в `boolean` дают `false`.

Как узнать, что это за значения? В спецификации ES5 в разделе 9.2 определена абстрактная операция `ToBoolean`, которая точно говорит, что происходит со всеми возможными значениями при попытке преобразования их в `boolean`.

Из этой таблицы мы получаем следующий список так называемых «ложных» значений:

- `undefined`
- `null`
- `false`
- `+0, -0` и `NaN`
- `""`

Вот и все. Если значение присутствует в списке, оно является «ложным», и при применении преобразования к логическому типу вы получите `false`.

Логика подсказывает, что, если значение не входит в этот список, оно должно входить в другой список, «истинных» значений. Но спецификация JS не определяет «истинный» список как таковой. Она содержит некоторые примеры (например, в ней явно указано, что все объекты являются истинными), но в основном спецификация просто подразумевает, что *все значения, не входящие в список ложных, являются истинными*.

Ложные объекты

Погодите минуту, название этого раздела выглядит странно. Я ведь *только что сказал*, что в спецификации все объекты названы истинными, верно? Не должно быть такой вещи, как «ложный объект».

Что это вообще такое?

Напрашивается предположение, что речь может идти об объектных обертках (см. главу 3) для ложных значений ("", 0 или `false`). Но это ловушка!

Посмотрите:

```
var a = new Boolean( false );
var b = new Number( 0 );
var c = new String( "" );
```

Мы знаем, что все три значения являются объектами (см. главу 3), в которых обернуты явно ложные значения. Но как будут интерпретироваться такие объекты: как `true` или как `false`? На этот вопрос легко ответить:

```
var d = Boolean( a && b && c );
d; // true
```

Итак, все три интерпретируются как `true`, потому что только в этом случае переменная `d` будет равна `true`.



Обратите внимание на обертку `Boolean(..)` вокруг выражения `a && b && c`. Зачем она здесь? Позднее мы еще вернемся к этому вопросу, так что просто запомните на будущее. А самые любопытные могут попробовать самостоятельно просчитать, что будет содержать `d`, если просто выполнить команду `d = a && b && c` без вызова `Boolean(..)`!

Итак, если «ложные объекты» — это не объекты, которые являются обертками для ложных значений, то чем еще они могут быть?

Тут проблема в том, что они могут встретиться в вашей JS-программе, при этом не являясь частью самого JavaScript.

Что-о-о?!

В некоторых ситуациях браузеры создают собственное *экзотическое* поведение для некоторых значений, а именно понятие «ложных объектов» поверх обычной семантики JS.

«Ложный объект» представляет собой значение, которое выглядит и работает как нормальный объект (свойства и все такое), но при попытке преобразования к `boolean` дает `false`.

Почему?!

Самый известный случай такого рода — `document.all`; этот объект (аналог массива) предоставляется вашей JS-программе JS-моделью DOM (не самим движком JS) и предоставляет элементы страницы вашей JS-программе. Когда-то он вел себя как обычный объект и действовал как истинное значение. Сейчас все изменилось. Объект `document.all` никогда не считался «стандартным» и уже давно считается устаревшим/не поддерживаемым.

«Тогда почему просто не удалить его?» Хорошая мысль, но увы. Они бы и сами этого хотели. Написано слишком много старого JS-кода, который зависит от этого объекта.

Тогда зачем заставлять его действовать как ложный? Потому что преобразования `document.all` в `boolean` (обычно в командах `if`) почти всегда используются как способ выявления старого, нестандартного IE. С тех пор IE уже давно пришел к соответствию стандартам и во многих случаях определяет направление развития Web ничуть не меньше, чем любой другой браузер.

Но весь старый код `if (document.all) { /* это IE */ }` все еще существует, и большая часть его, вероятно, никогда не уйдет. И весь старый код все еще предполагает, что он работает в IE десятилетней давности, что оставит не лучшие впечатления для пользователей IE.

Таким образом, `document.all` нельзя полностью удалить, но IE не хочет, чтобы код `if (document.all) { .. }` работал — это нужно

для того, чтобы пользователи современных версий IE получали новую, соответствующую стандартам логику кода.

«Что же делать?»

«Придумал! Давайте изуродуем систему типов JS и сделаем вид, что `document.all` является ложным!»

Полный отстой. Это одна из тех безумных проблем, совершенно непонятных для многих разработчиков JS. Впрочем, альтернатива (не делать ничего в безысходной ситуации) еще хуже.

Итак, мы имеем то, что имеем: безумные, нестандартные «ложные объекты», которые добавляются в JavaScript браузерами.

Истинные значения

Но вернемся к списку истинности. Какие же значения относятся к категории истинных? Помните: значение считается истинным, если оно не входит в список ложных.

Пример:

```
var a = "false";
var b = "0";
var c = '';
var d = Boolean( a && b && c );
d;
```

Как вы думаете, какое значение примет `d`? Вариантов всего два: `true` или `false`.

`True`. Почему? Потому что, несмотря на то что содержимое этих строковых значений выглядит как ложные значения, сами строковые значения все истинны, так как `""` – единственное строковое значение в ложном списке.

А как в этом случае?

```
var a = [] ;           // пустой массив -- истинный или ложный?  
var b = {} ;          // пустой объект -- истинный или ложный?  
var c = function(){ } ; // пустая функция -- истинная или ложная?  
var d = Boolean( a && b && c );  
  
d;
```

Да, вы догадались правильно — `d` все равно содержит `true`. Почему? По той же причине, что и прежде. Что бы там ни казалось, `[]`, `{}` и `function(){}()` *не* входят в ложный список, а следовательно, являются истинными значениями.

Другими словами, список истинных значений получается бесконечно длинным. Его вообще невозможно построить. Можно только создать конечный ложный список и обращаться к нему.

Не пожалейте пяти минут, запишите ложный список на стикере и прикрепите его к своему монитору — или запомните, если вам так удобнее. В любом случае вы сможете легко построить такой виртуальный истинный список каждый раз, когда потребуется. Для этого достаточно просто определить, входит значение в ложный список или нет.

Истинные и ложные значения важны прежде всего для понимания того, как значение поведет себя при преобразовании (явном или неявном) в значение `boolean`. Итак, теперь вы представляете себе эти два списка, и мы можем перейти к примерам преобразования типов.

Явное преобразование типов

Под явным преобразованием типов понимаются преобразования типов, очевидно и явно выраженные в программе. Существует обширный класс применений преобразований типов, которые

очевидно относятся к категории явных преобразований для большинства разработчиков.

Здесь мы постараемся выявить в коде закономерности, позволяющие четко и очевидно показать, что значение преобразуется из одного типа в другой, чтобы не создавать потенциальных проблем для будущих разработчиков. Чем однозначнее выражаются намерения, тем больше вероятность того, что кто-то в будущем прочитает ваш код и без лишних усилий поймет, что вы имели в виду.

По поводу явных преобразований нет особых расхождений во мнениях, так как оно чрезвычайно близко соответствует общепринятой практике преобразований в языках со статической типизацией. Соответственно мы примем как данность (пока), что явное преобразование не считается чем-то нехорошим или неоднозначным. Впрочем, позднее мы еще вернемся к этому вопросу.

Явные преобразования: `String <--> Number`

Мы начнем с простейшей и, пожалуй, самой распространенной операции преобразования типов: преобразование значений между строковым и числовым представлением.

Для явного преобразования между строками и числами используются встроенные функции `String(..)` и `Number(..)` (которые мы называли «встроенными конструкторами» в главе 3), но что *очень важно* – перед ними не ставится ключевое слово `new`. А это означает, что мы не создаем объектные обертки.

Вместо этого выполняется явное преобразование между двумя типами:

```
var a = 42;  
var b = String( a );  
  
var c = "3.14";
```

```
var d = Number( c );
b; // "42"
d; // 3.14
```

`String(..)` преобразует любое другое значение в примитивное значение `string` по правилам операции `ToString`, упоминавшейся выше. `Number(..)` преобразует любое другое значение в примитивное значение `number` по правилам операции `ToNumber`, упоминавшейся выше. Я называю это явным преобразованием, поскольку обычно большинству разработчиков вполне очевидно, что конечным результатом этих операций является преобразование типа.

Собственно, этот вариант использования выглядит практически так же, как в других языках со статической типизацией.

Например, в C/C++ можно использовать запись `(int)x` или `int(x)` — оба варианта преобразуют значение из переменной `x` в целое число. Обе формы разрешены, но многие предпочитают вторую, похожую на вызов функции. В JavaScript запись `Number(x)` выглядит очень похоже. Важно ли, что в JS это *действительно* вызов функции? В общем-то, нет.

Кроме `String(..)` и `Number(..)`, существуют и другие способы «явного» преобразования этих значений между строками и числами:

```
var a = 42;
var b = a.toString();

var c = "3.14";
var d = +c;

b; // "42"
d; // 3.14
```

Вызов `a.toString()` формально можно считать явным (понятно, что `toString` означает «в строку»), но здесь также кроется не-

которая скрытая неявность. `ToString()` не может вызываться для *примитивных* значений, таких как 42. По этой причине JS автоматически «упаковывает» (см. главу 3) 42 в объектную обертку, чтобы для объекта можно было вызвать `toString()`. Другими словами, этот вызов можно назвать «явно-неявным».

`+c` здесь демонстрирует *унарную форму* (оператор только с одним операндом) оператора `+`. Вместо выполнения математического сложения (или конкатенации строк – см. ниже) унарный `+` явно преобразует свой operand (`c`) в числовое значение.

Является ли `+c` явным преобразованием? Это зависит от вашего опыта и точки зрения. Если вы знаете (а сейчас вы уже знаете), что унарный `+` явно предназначен для преобразования чисел, то все очевидно и явно. Тем не менее если вы никогда не видели его ранее, это покажется ужасно запутанным, неявным, чреватым скрытыми побочными эффектами и т. д.



В сообществе разработчиков JS с открытым кодом унарный `+` считается общепризнанной формой явного преобразования.

Даже если форма `+c` вам нравится, безусловно есть места, в которых она выглядит ужасно запутанной. Пример:

```
var c = "3.14";
var d = 5 + +c;

d; // 8.14
```

Унарный оператор `-` тоже выполняет преобразование, как и `+`, но он также меняет знак числа. Тем не менее нельзя поставить два минуса (--) рядом друг с другом для восстановления знака, поскольку такие символы будут интерпретированы как оператор декремента. Вместо этого нужно поставить между ними пробел: `- - "3.14";` это приведет к преобразованию к `3.14`.

Вероятно, наряду с унарной формой оператора вы сможете придумать всевозможные комбинации бинарных операторов (по аналогии с `+` для сложения). Другой безумный пример:

```
1 + - + + - + 1; // 2
```

Возможно, вам стоит серьезно подумать о том, чтобы отказаться от унарных преобразований `+` (или `-`), располагающихся вплотную к другим операторам. Хотя такие решения работают, они почти всегда нежелательны. Даже `d = +c` (или `d =+ c`, если уж на то пошло!) слишком легко спутать с `d += c`, а это совсем другое!



Другое в высшей степени запутанное применение унарного `+` — когда этот символ располагается рядом с оператором инкремента `++` или оператором декремента `--`. Примеры такого рода: `a +++b`, `a + ++b` и `a + + +b`. За дополнительной информацией о `++` обращайтесь к разделу «Побочные эффекты выражений».

Помните: мы стараемся явно выразить намерения и уменьшить путаницу, а не усугублять ситуацию!

Преобразование даты в число

Другим распространенным использованием унарного оператора `+` является преобразование объекта `Date` в число. Результат представляет собой временной штамп UNIX (количество миллисекунд, прошедших с 1 января 1970 00:00:00 UTC) для представления значений даты/времени:

```
var d = new Date( "Mon, 18 Aug 2014 08:53:06 CDT" );
+d; // 1408369986000
```

Чаще всего эта идиома применяется для получения текущего момента времени в виде временного штампа:

```
var timestamp = +new Date();
```



Некоторые разработчики знают об особенном синтаксическом трюке в JavaScript: круглые скобки () для вызова конструктора (функция, вызываемая с new) *необязательны* только в том случае, если при вызове не передаются аргументы. Таким образом, вы можете встретить форму `var timestamp = +new Date();`. Тем не менее не все разработчики согласны с тем, что отсутствие () улучшает удобочитаемость, поскольку это необычное синтаксическое исключение относится только к форме вызова `new fn()`, а не к обычной форме вызова `fn()`.

Однако преобразование типа – не единственный способ получения временного штампа из объекта Date. Решение без преобразования даже лучше, поскольку оно получается более явным:

```
var timestamp = new Date().getTime();
// var timestamp = (new Date()).getTime();
// var timestamp = (new Date).getTime();
```

Но *еще лучше* воспользоваться статической функцией `Date.now()`, добавленной в ES5:

```
var timestamp = Date.now();
```

А если вы хотите определить полифил `Date.now()` для старых браузеров, это делается просто:

```
if (!Date.now) {
  Date.now = function() {
    return +new Date();
  };
}
```

Я рекомендую держаться подальше от форм преобразования, связанных с датами. Используйте `Date.now()` для текущих временных штампов и `new Date(...).getTime()` для получения временного штампа конкретной (не текущей) даты/времени, которые нужно задать в программе.

Занятный случай с оператором ~

Один преобразующий оператор JS, о котором часто забывают и который обычно создает изрядную путаницу, — оператор ~ (тильда), он же «поразрядное NOT». Даже многие из тех, кто понимает, как он работает, предпочитают держаться от него по-дальше. Мы же смело разберемся здесь с оператором ~ и посмотрим, можно ли с него стрясти что-то полезное.

В разделе «32-разрядные целые числа (со знаком)» я показал, что поразрядные операторы в JS определены только для 32-разрядных операций; это означает, что их операнды преобразуются для соответствия представлений в виде 32-разрядных величин. Правила этого преобразования определяются абстрактной операцией `ToInt32` (спецификация ES5, раздел 9.5).

`ToInt32` сначала выполняет преобразование `ToNumber`; это означает, что значение "123" сначала будет преобразовано в 123 до применения правил `ToInt32`.

Хотя использование поразрядных операторов (таких, как `|` или `~`) формально не является преобразованием типа (поскольку тип не изменяется), с некоторыми специальными числовыми значениями он дает эффект, приводящий к изменению значения числа.

Например, рассмотрим оператор «поразрядного OR» из идиомы «пустой операции» `0 | x`, которая (как было показано в главе 2) фактически только выполняет преобразование `ToInt32`:

```
0 | -0;          // 0
0 | NaN;         // 0
0 | Infinity;   // 0
0 | -Infinity;  // 0
```

Эти специальные числа непредставимы в 32-разрядном формате (поскольку они происходят из 64-разрядного стандарта IEEE 754 — см. главу 2), поэтому `ToInt32` просто возвращает `0` как результат для таких значений.

Вопрос о том, является ли форма `0 | __` явной формой, преобразующей операции `ToInt32`, или же она скорее относится к неявным, остается спорным. С точки зрения спецификации она бесспорно должна считаться явной, но если вы не понимаете поразрядные операции на таком уровне, результат может показаться неявным, если не волшебным. Тем не менее в соответствии с другими предположениями этой главы, мы будем считать его явным.

Итак, обратимся к `~`. Оператор `~` сначала выполняет «преобразование» к 32-разрядному числовому значению, а затем выполняет поразрядное отрицание (с переключением каждого бита).



Это очень похоже на то, как оператор `!` не только преобразует свое значение к `boolean`, но и переключает его в противоположное состояние (см. описание унарного оператора `!` в разделе «Явные преобразования: * --> Boolean»).

Но... зачем?! Зачем нам заниматься переключением битов? Все это сильно специализированные возможности, не предназначенные для повседневного использования. Разработчикам JS редко приходится думать об отдельных битах.

Другое определение `~` происходит из классической дискретной математики: `~` вычисляет дополнительный двоичный код. Здорово! Спасибо, теперь все встало на свои места!

Пробуем еще раз: `~x` — примерно то же, что `-(x+1)`. Это странно, но несколько упрощает рассуждения. Итак:

```
~42; // -(42+1) ==> -43
```

И к чему все эти рассуждения о `~`, какое отношение они имеют к преобразованию типов? Перехожу к делу.

Возьмем `-(x+1)`. С каким единственным значением можно выполнить эту операцию, чтобы получить результат `0` (или с технической точки зрения `-0`)? `-1`. Иначе говоря, в диапазоне чис-

ловых значений `~` будет выдавать ложное (легко преобразуемое к `false`) значение `0` для входного значения `-1` и другие истинные числа во всех остальных случаях.

Почему это важно?

`-1` часто называется «сигнальным значением», что, по сути, означает значение, имеющее особый семантический смысл в более широком спектре значений того же типа (числа). Язык С использует сигнальные значения `-1` для многих функций, возвращающих значения `>=0` для «успеха» и `-1` для «неудачи».

JavaScript следует этому принципу при определении строковой операции `indexOf(..)`, которая ищет вхождение подстроки и возвращает индекс его начальной позиции (начиная с нуля); если вхождение не найдено, возвращается `-1`.

`IndexOf(..)` достаточно часто используется не только для получения позиции, но и для логической проверки присутствия/отсутствия подстроки в другой строке. Обычно разработчики выполняют такие проверки так:

```
var a = "Hello World";  
  
if (a.indexOf( "lo" ) >= 0) {    // true  
    // найдено!  
}  
if (a.indexOf( "lo" ) != -1) {    // true  
    // найдено  
}  
if (a.indexOf( "ol" ) < 0) {    // true  
    // не найдено!  
}  
if (a.indexOf( "ol" ) == -1) {    // true  
    // не найдено!  
}
```

На мой взгляд, все эти `>= 0` и `== -1` выглядят довольно уродливо. Это так называемая «дырявая абстракция», через которую в код «вытекает» поведение нижележащей реализации, то есть ис-

пользование сигнального значения `-1` как признака «неудачи». Я предполагаю скрывать такие подробности.

И наконец-то становится ясно, чем `~` может нам помочь! Использование `~` с `indexOf()` «преобразует» значение к виду, который нормально приводится к `boolean`:

```
var a = 'Hello World';
~a.indexOf( "lo" );           // -4    <-- truthy!
if (~a.indexOf( "lo" )) {    // true
    // найдено!
}
~a.indexOf( "ol" );           // 0    <-- falsy!
!~a.indexOf( "ol" );          // true
if (!~a.indexOf( "ol" )) {   // true
    // не найдено!
}
```

`~` берет возвращаемое значение `indexOf(..)` и преобразует его: для признака «неудачи» `-1` мы получаем ложное значение `0`, тогда как все остальные значения истинны.



Псевдоалгоритм `- (x+1)` для `~` подразумевает, что результат `-1` равен `-0`, но на самом деле вы получите `0`, потому что фактически выполняется поразрядная, а не математическая операция.

Технически `if (~a.indexOf(..))` все еще полагается на неявное преобразование полученного результата `0` в `false`, или ненулевого значения в `true`. Но в целом мне кажется, что прием с `~` все еще больше похож на механизм явного преобразования — лишь бы вы знали, что должно делаться с этой идиомой.

Я считаю, что такой код выглядит намного понятнее, чем предыдущая мешаница с `>= 0 / == -1`.

Усечение битов

Есть еще одно место, в котором вы можете столкнуться с `~` в коде: некоторые разработчики используют `~~` (двойная тильда) для усечения дробной части числа (то есть его «преобразования» к целому числу). Часто (хотя и ошибочно) считается, что этот результат эквивалентен вызову `Math.floor(..)`.

Как же работает `~~`? Сначала `~` применяет «преобразование» `ToInt32` и поразрядно изменяет биты, после чего второй оператор `~` осуществляет другое поразрядное переключение, возвращая все биты в исходное состояние. В итоге вы получите просто «преобразование» `ToInt32` (то есть усечение).



Поразрядное двойное отрицание, или `~~`, очень похоже на поведение `!!`, объясняемое в разделе «Явные преобразования: `* --> Boolean`».

Однако `~~` нуждается в некоторых пояснениях. Во-первых, эта конструкция надежно работает для 32-разрядных значений. Но, что важнее, с отрицательными числами она работает не так, как `Math.floor(..)!`

```
Math.floor( -49.6 );      // -50
~~-49.6;                  // -49
```

Если не считать различий с `Math.floor(..)`, `~~x` может выполнять усечение к (32-разрядному) целому числу. Но то же самое делает и `x | 0`, и вроде бы с (немного) меньшими усилиями.

Итак, когда же стоит выбирать `~~x` вместо `x | 0`? Приоритет операторов (см. главу 5):

```
~~1E20 / 10;            // 166199296
1E20 | 0 / 10;          // 1661992960
(1E20 | 0) / 10;        // 166199296
```

В данном случае действует то же правило, что и для всех остальных приведенных советов: используйте `~` и `~~` как явный механизм преобразования только в том случае, если каждый разработчик, который будет читать/писать такой код, хорошо понимает принцип работы этих операторов!

Явные преобразования: разбор числовых строк

Аналогичного результата при преобразовании строки в число можно достигнуть разбором числа по содержимому строки. Тем не менее между этим вариантом разбора и преобразованием типов, рассмотренным выше, существуют определенные различия.

Пример:

```
var a = "42";
var b = "42px";

Number( a );      // 42
parseInt( a );   // 42

Number( b );      // NaN
parseInt( b );   // 42
```

Разбор числового значения из строки *устойчив* к нечисловым символам (при обнаружении такого символа просто прекращается разбор слева направо), тогда как при преобразовании типа при обнаружении нечислового символа происходит *сбой*, а результатом операции является значение `NaN`. Разбор строки не должен рассматриваться как замена преобразования типа. Эти две задачи при всем внешнем сходстве служат разным целям. Применяйте разбор строк в тех случаях, когда вас не интересуют другие нечисловые символы, которые могут присутствовать в правой части. Преобразуйте строку (в число), когда допустимы только числовые значения, а строки вида "42px" должны отвергаться.

Не забудьте, что `parseInt(..)` работает со строковыми значениями. Передавать `parseInt(..)` число полностью бессмысленно. Также бессмысленно передавать значения любых других типов: `true`, `function(){..}`, `[1,2,3]` и т. д.



У `parseInt(..)` существует парная операция `parseFloat(..)`, которая (как нетрудно догадаться по имени) разбирает строку в число с плавающей точкой.

При передаче значения, не являющегося строкой, переданное значение будет сначала автоматически преобразовано в строку (см. раздел «`ToString`»), что очевидным образом должно рассматриваться как скрытое неявное преобразование. Полагаться на такое поведение в программе не рекомендуется, поэтому никогда не используйте `parseInt(..)` с нестроковыми значениями.

До ES5 у `parseInt(..)` существовала другая скрытая проблема, которая служила источником многих ошибок в программах JS. Если не передать второй аргумент для обозначения основания системы счисления, в которой должно интерпретироваться содержимое числовой строки, `parseInt(..)` будет пытаться угадать систему счисления по первому символу.

Если первым символом является `x` или `X`, `parseInt(..)` (по соглашению) считает, что строка должна интерпретироваться как шестнадцатеричное число (основание 16). Если первым символом является `0`, `parseInt(..)` (опять-таки по соглашению) считает, что строка должна интерпретироваться как восьмеричное число (основание 8). Шестнадцатеричные строки (с начальным символом `x` или `X`) не так легко перепутать с десятичными. Тем не менее с восьмеричными числами такая путаница возникала невероятно часто. Пример:

```
var hour = parseInt( selectedHour.value );
var minute = parseInt( selectedMinute.value );
```

```
console.log(  
  "The time you selected was: " + hour + ":" + minute  
)
```

Выглядит безобидно, не правда ли? Попробуйте выбрать для `hour` значение `08`, а для `minute` значение `09`. Вы получите `0:0`. Почему? Потому что ни `8`, ни `9` не являются допустимыми символами в восьмеричной системе счисления.

До ES5 проблема решалась просто, хотя об этом часто забывали: *всегда передавайте 10 во втором аргументе*. Такой вызов был совершенно безопасным:

```
var hour = parseInt( selectedHour.value, 10 );  
var minute = parseInt( selectedMinute.value, 10 );
```

Начиная с ES5 `parseInt(..)` уже не пытается гадать. Если не указать другое, предполагается десятичная система. Уже намного лучше. Главное, будьте осторожны в том случае, если ваш код должен работать в средах до ES5. Тогда следует передавать `10` во втором аргументе.

Передача нестроковых значений

Один из известных примеров странного поведения `parseInt(..)` встречается в одном саркастическом шуточном посте, издавающем над этим поведением JS:

```
parseInt( 1/0, 19 ); // 18
```

Распространенное (но совершенно ошибочное) предположение: «Если передать `Infinity` и попытаться выделить целое число, я получу `Infinity`, а не `18`». JS совсем рехнулся, скажете?

Хотя пример искусственный и нереальный, на время забудем об этом и разберемся, действительно ли JS настолько безумен.

Прежде всего, самый очевидный грех — передача нестрокового значения `parseInt(..)`. Это грубейшая ошибка. Сделав это, вы

сами напрашивается на неприятности. Но даже если вы действительно напрашиваетесь, JS вежливо преобразует переданное значение в строку, которую затем попытается разобрать.

Кто-то скажет, что это неразумное поведение, и функция `parseInt(..)` должна отказаться от работы с нестроковыми значениями. Должна ли она инициировать ошибку? Откровенно говоря, это будет в духе Java. Меня ужасает сама мысль о том, что JS начнет швыряться ошибками направо и налево, и почти каждую строку придется заключать в `try..catch`.

Должна ли функция вернуть `NaN`? Возможно. Но как насчет такого:

```
parseInt( new String( "42" ) );
```

Должен ли и этот вызов завершиться неудачей? Это нестроковое значение. Если вы хотите, чтобы объектная обертка `String` была распакована в "42", так ли необычно для 42 сначала превратиться в "42", чтобы вы снова получили 42 в результате обратного разбора?

Я бы возразил, что это наполовину явное, наполовину неявное преобразование может оказаться очень полезным. Пример:

```
var a = {
  num: 21,
  toString: function() { return String( this.num * 2 ); }
};

parseInt( a ); // 42
```

Тот факт, что `parseInt(..)` принудительно преобразует свое значение в строку для выполнения разбора, на самом деле вполне разумен. Если вы подаете мусор на вход и получаете мусор на выходе, не вините мусорное ведро — оно просто честно выполнило свою работу.

Итак, если передать такое значение, как `Infinity` (очевидный результат `1/0`), какое строковое представление будет наиболее подходящим для такого преобразования? В голову приходят

только два разумных варианта: "Infinity" и " ∞ ". JS выбирает "Infinity". И я думаю, совершенно правильно.

Я считаю, хорошо, что все значения в JS обладают неким стро-ковым представлением по умолчанию. Иначе они превратились бы в загадочные «черные ящики», которые делали бы невозмож-ной отладку и разумный анализ поведения.

А как насчет основания 19? Разумеется, полная нелепость. Ни одна реальная JS-программа не станет использовать девятнад-цатеричную систему счисления. Да, это абсурдно. Но давайте на время забудем об этом. В девятнадцатеричной системе счисления допустимыми цифровыми символами являются 0-9 и a-i (в лю-бом регистре).

Вернемся к примеру `parseInt(1/0, 19)`. По сути это вызов `parseInt("Infinity", 19)`. Как происходит разбор строки? Первый символ "I" соответствует значению 18 в абсурдной де-вятнадцатеричной системе. Второй символ "n" не входит в до-пустимый набор цифровых символов, поэтому разбор спокойно останавливается, подобно тому как он останавливался при об-наружении "p" в "42px". Результат? 18. Все абсолютно логично. Аспекты поведения, которые приводят нас именно к такому результату, а не к ошибке или к `Infinity`, очень важны для JS, и так легко отбросить их не удастся.

Другие примеры такого поведения с `parseInt(..)`, которое может показаться удивительным, но на самом деле вполне разумно:

```
parseInt( 0.000008 );      // 0   ("0" от "0.000008")
parseInt( 0.0000008 );     // 8   ("8" от "8e-7")
parseInt( false, 16 );      // 250 ("fa" от "false")
parseInt( parseInt, 16 );   // 15  ("f" от "function..")

parseInt( "0x10" );        // 16
parseInt( "103", 2 );      // 2
```

На самом деле поведение `parseInt(..)` последовательно и впол-не прогнозируемо. Если использовать эту функцию правильно,

вы получите разумные результаты. Если использовать ее неправильно, то получите очень странные результаты, но виноват в этом будет не JavaScript.

Явные преобразования: * --> Boolean

Теперь посмотрим, как перевести любое нелогическое значение в `boolean`.

Как и в случае со `String(..)` и `Number(..)`, `Boolean(..)` (без `new`, конечно!) позволяет явно выразить принудительное преобразование к `ToBoolean`:

```
var a = "0";
var b = [];
var c = {};

var d = "";
var e = 0;
var f = null;
var g;

Boolean( a ); // true
Boolean( b ); // true
Boolean( c ); // true

Boolean( d ); // false
Boolean( e ); // false
Boolean( f ); // false
Boolean( g ); // false
```

Хотя форма `Boolean(..)`, безусловно, выражена явно, она не так уж распространена и не идиоматична.

Подобно тому как унарный оператор `+` преобразует значение в число (см. выше), унарный оператор отрицания `!` явно преобразует значение в `boolean`. *Проблема* в том, что значение при этом переходит из истинного в ложное и наоборот. Итак, для преобразования в `boolean` разработчики JS чаще всего используют

оператор двойного отрицания `!!`, потому что второй `!` возвращает разряд в исходное состояние:

```
var a = "0";
var b = [];
var c = {};
var d = "";
var e = 0;
var f = null;
var g;

!!a;    // true
!!b;    // true
!!c;    // true

!!d;    // false
!!e;    // false
!!f;    // false
!!g;    // false
```

Все эти преобразования к `ToBoolean` будут выполняться неявно без `Boolean(..)` или `!!` при использовании в логическом контексте (например, в команде `if (..) ..`). Но здесь явное приведение к `boolean` нужно для того, чтобы более четко выразить намеренный характер преобразования к `ToBoolean`.

Другой типичный пример явного преобразования к `ToBoolean` встречается тогда, когда вы хотите выполнить принудительное преобразование `true/false` в сериализации JSON структуры данных:

```
var a = [
  1,
  function(){ /*..*/ },
  2,
  function(){ /*..*/ }
];

JSON.stringify( a ); // "[1,null,2,null]"

JSON.stringify( a, function(key,val){
  if (typeof val == "function") {
```

```
// выполнить преобразование функции в `ToBoolean`  
    return !val;  
}  
else {  
    return val;  
}  
};  
// "[1,true,2,true]"
```

Если вы пришли к JavaScript из Java, следующая идиома может показаться знакомой:

```
var a = 42;  
  
var b = a ? true : false;
```

Тернарный оператор `? :` проверяет `a` на истинность, и на основании результатов присваивает `b` `true` или `false` соответственно.

На поверхности эта идиома выглядит как разновидность явного преобразования к типу `ToBoolean`, поскольку очевидно, что результатом этой операции может быть только значение `true` или `false`.

Однако здесь присутствует скрытое неявное преобразование: выражение `a` сначала должно быть преобразовано к `boolean` для проведения проверки на истинность. Я рекомендую полностью избегать этой идиомы в JavaScript. Она не приносит никакой реальной пользы и что еще хуже — маскируется под нечто такое, чем она не является.

`Boolean(a)` и `!a` являются лучшими способами явного преобразования.

Неявное преобразование

Неявным преобразованием называется скрытое преобразование с неочевидными побочными эффектами, неявно происходящими в результате других действий. Иначе говоря, к категории неявных

относятся любые преобразования, которые не очевидны при чтении кода (для вас).

Хотя цель явных преобразований совершенно понятна (сделать код более явным и понятным), неявные преобразования добиваются противоположной цели: усложняют понимание кода.

Вероятно, именно здесь кроется источник большей части недовольства преобразованиями типов. Большинство жалоб на «преобразования типов JavaScript» на самом деле относится (независимо от того, понимают это их авторы или нет) к неявным преобразованиям.



Дуглас Крокфорд, автор книги «*JavaScript: the Good Parts*» (Крокфорд Д. *JavaScript: сильные стороны*. — СПб.: Питер, 2013. — 176 с.: ил. — (Серия «Бестселлеры O'Reilly»).), в выступлениях и статьях говорил о том, что преобразования типов JavaScript следует избегать. Но, похоже, он имел в виду, что плохи неявные преобразования (по его мнению). Однако если почитать его собственный код, вы найдете в нем множество примеров преобразований — как неявных, так и явных! Откровенно говоря, его гнев в первую очередь направлен против операции `==`, но как вы вскоре увидите в этой главе, это лишь часть механизма преобразования.

Итак, неявные преобразования типов вредны? Они опасны? В архитектуре JavaScript существует дефект? Их следует избегать любой ценой?

Уверен, что многие читатели склонны с энтузиазмом воскликнуть: «Да!»

Не торопитесь. Сначала выслушайте меня.

Попробуйте взглянуть на неявные преобразования и то, чем они могут быть, с другой стороны — не рассматривайте их как «противоположность для хороших явных преобразований». Такая точка зрения слишком узка, она упускает один важный нюанс.

Определим цель неявного преобразования иначе: неявное преобразование устраняет излишества, шаблонный код и/или не нужные подробности реализации, которые только загромождают код шумом, отвлекающим от более важных намерений разработчика.

Неявное упрощение

Прежде чем браться за JavaScript, покажу вам некий псевдокод из вымышленного языка с сильной типизацией:

```
SomeType x = SomeType( AnotherType( y ) )
```

В этом примере `y` содержит значение произвольного типа, которое нужно преобразовать к типу `SomeType`. Проблема в том, что язык не может напрямую перейти от текущего содержимого `y` к `SomeType`. Ему нужен промежуточный шаг: сначала значение преобразуется к `AnotherType`, а затем из `AnotherType` в `SomeType`.

А теперь представьте, что язык (или определение, которое вы можете сами создать в языке) *позволит* вам использовать запись:

```
SomeType x = SomeType( y )
```

Разве вы не согласны с тем, что мы упростили преобразование типа, для того чтобы избавиться от лишнего «шума» на промежуточном шаге преобразования? Я имею в виду, неужели именно в этой точке кода так важно знать, что у сначала преобразуется к `AnotherType`, перед тем как перейти к `SomeType`?

Кто-то скажет, что в некоторых обстоятельствах это важно. Но я думаю, что аналогичный аргумент можно привести для многих других ситуаций, в которых упрощение *действительно упрощает чтение кода* за счет абстрагирования или сокрытия многих подробностей — либо в самом языке, либо в его собственных абстракциях.

Несомненно, где-то за кулисами промежуточное преобразование все равно выполняется. Но если эта подробность скрыта из виду, мы можем просто рассматривать переход у к типу *SomeType* как обобщенную операцию и спрятать все хлопотные подробности.

Пусть аналогия не идеальна, в оставшейся части этой главы я хочу показать: неявные преобразования JS могут рассматриваться как механизм, предоставляющий аналогичную возможность для вашего кода.

Тем не менее — и это очень важно! — это утверждение не является абсолютным и однозначно истинным. Вокруг неявного преобразования блуждает множество скрытых опасностей, от которых вашему коду будет гораздо больше вреда, чем пользы от любых потенциальных улучшений удобочитаемости. Очевидно, вам придется научиться избегать таких конструкций, чтобы не заражать код всевозможными ошибками.

Многие разработчики полагают, что, если механизм способен сделать полезную вещь *A*, но из-за злоупотреблений или неправильного использования он может сотворить ужасную вещь *Z*, от этого механизма стоит отказаться просто ради безопасности. Я считаю иначе: не останавливайтесь на этом. «Не выплескивайте ребенка вместе с водой!» Не стоит полагать, что неявное преобразование всегда плохо, потому что вы не заметили ничего, кроме «плохих» частей. Я считаю, что «хорошие» части тоже существуют, и теперь хочу помочь вам найти их и взять на вооружение!

Неявные преобразования: *String <--> Number*

Ранее в этой главе исследовались явные преобразования между строковыми и числовыми значениями. Теперь рассмотрим ту же задачу, но с неявным преобразованием. Но сначала необходимо изучить некоторые нюансы операций, требующих неявного преобразования.

Оператор `+` перегружен, чтобы он мог использоваться как для числового сложения, так и для конкатенации строк. Как JS определит, какой тип операции вы хотите использовать? Пример:

```
var a = "42";
var b = "0";

var c = 42;
var d = 0;

a + b; // "420"
c + d; // 42
```

Чем отличаются эти два сложения, если в одном случае получается `"420"`, а в другом `42`? Согласно распространенному заблуждению, тем, что один операнд (или оба) является строкой, из-за чего `+` выбирает конкатенацию строк. Отчасти справедливо, но все не так просто.

Пример:

```
var a = [1,2];
var b = [3,4];

a + b; // "1,23,4"
```

Ни один из этих операндов не является строкой, но очевидно, что они оба преобразуются в строки, после чего вступает в силу конкатенация строк. Что же здесь происходит на самом деле?



Дальше идут скучные и сугубо технические обсуждения спецификации. Если они вас пугают, пропустите следующие два абзаца!

Согласно спецификации ES5, раздел 11.6.1, алгоритм `+` (когда операндом является значение `object`) выполняет конкатенацию, если хотя бы один операнд уже является строкой либо следующая

последовательность действий дает строковое представление. Когда алгоритм `+` получает объект (в том числе и массив) в любом из операндов, он сначала вызывает абстрактную операцию `ToPrimitive` (раздел 9.1) для этого значения, а затем вызывает алгоритм `[[DefaultValue]]` (раздел 8.12.8) с контекстным указанием `number`.

Если вы внимательно следили за происходящим, то заметили, что эта операция теперь идентична тому, как абстрактная операция `ToNumber` работает с объектами (см. «`ToNumber`»). Операция `valueOf()` для массива не сможет выдать простой примитив, поэтому потом она перейдет к представлению `toString()`. Таким образом, два массива превращаются в `"1,2"` и `"3,4"` соответственно. После этого `+` выполняет конкатенацию двух строк, как обычно: `"1,23,4"`.

Отложим все эти подробности и вернемся к более раннему, упрощенному объяснению: если один из операндов `+` является строкой (или становится ею в результате описанных выше действий), то выполняется операция конкатенации строк. В остальных случаях всегда выполняется числовое сложение.



Одна из часто упоминаемых странностей преобразования типов — `[] + {}` и `{ } + []`. Эти два выражения дают соответственно `"[object Object]"` и `0`. Впрочем, это еще не все; подробности будут рассмотрены в разделе «Блоки».

Что это означает для неявного преобразования?

Число можно преобразовать в строку простым «сложением» его с пустой строкой:

```
var a = 42;  
var b = a + "";  
  
b; // "42"
```



Числовое сложение с оператором `+` обладает свойством коммутативности; это означает, что `2 + 3` — то же самое, что `3 + 2`. Конкатенация строк `+` очевидно не является коммутативной, но в частном случае `""` она фактически коммутативна, поскольку `a + ""` и `"" + a` дают один результат.

Преобразование чисел в строки операцией `a + ""` в высшей степени распространено и идиоматично. Интересно, что даже некоторые из самых яростных критиков неявного преобразования продолжают использовать этот прием в своем коде (вместо одной из явных альтернатив).

Я считаю, что это отличный пример полезной формы неявного преобразования, как бы ни критиковали этот механизм!

При сравнении этого неявного преобразования `a + ""` с нашим предыдущим примером явного преобразования `String(a)` встречается еще одна дополнительная странность, о которой необходимо знать. Из-за особенностей работы абстрактной операции `ToPrimitive` `a + ""` вызывает для значения `a` метод `valueOf()`, возвращаемое значение которого преобразуется в `string` внутренней абстрактной операцией `ToString`. Однако `String(a)` просто вызывает `toString()` напрямую.

Оба способа в итоге дают строку, но если использовать объект вместо обычного примитивного числа, вы не всегда получите *то же* строковое значение!

Пример:

```
var a = {
  valueOf: function() { return 42; },
  toString: function() { return 4; }
};

a + "";           // "42"

String( a );    // "4"
```

Обычно такие проблемы не мучают вас, если только вы не пытаетесь создавать какие-то запутанные структуры данных и операции, однако вы должны быть осторожны при определении собственных методов `valueOf()` и `toString()` для одного объекта, так как конкретный способ преобразования может повлиять на результат.

А как насчет другого направления? Как невидимо преобразовать строку в число?

```
var a = "3.14";
var b = a - 0;

b; // 3.14
```

Оператор `-` определен только для числового вычитания, так что `a - 0` инициирует преобразование значения `a` в число. Хотя запись `a * 1` или `a / 1` встречается намного реже, она приводит к тому же результату, так как эти операции тоже определены только для числовых операций.

Как насчет использования объектных значений с оператором `-`? Происходит то же, что с `+` в предыдущем примере:

```
var a = [3];
var b = [1];

a - b; // 2
```

Значения-массивы должны стать числами, но они сначала будут преобразованы в строки (с использованием сериализации `toString()`, как и ожидалось). Эти строки затем преобразуются в числа, с которыми будет выполнено вычитание `-`.

Невидимое преобразование строк и чисел и есть то самое скрытое зло, о котором вы слышали столько страшных историй? Я так не думаю. Сравните `b = String(a)` (явное) с `b = a + ""` (невидимое). Можно привести примеры, в которых оба подхода могут быть полезными в вашем коде.

Несомненно, запись `b = a + ""` чаще встречается в JS-программах и обладает самостоятельной полезностью, как бы вы ни относились к достоинствам и недостаткам неявного преобразования в целом.

Неявные преобразования: `Boolean --> Number`

Пожалуй, неявные преобразования по-настоящему проявляют себя при упрощении некоторых типов сложных логических операций в простое числовое сложение. Конечно, это не прием общего назначения, а конкретное решение для конкретных случаев.

Пример:

```
function onlyOne(a,b,c) {  
    return !!((a && !b && !c) ||  
              (!a && b && !c) || (!a && !b && c));  
}  
  
var a = true;  
var b = false;  
  
onlyOne( a, b, b ); // true  
onlyOne( b, a, b ); // true  
  
onlyOne( a, b, a ); // false
```

Функция `onlyOne(..)` должна возвращать `true` только в том случае, если ровно один из ее аргументов равен `true` (или является истинным). Для получения итогового *возвращаемого* значения она использует неявное преобразование для проверок истинности и явное преобразование в других случаях.

А если эта функция должна аналогичным образом обрабатывать четыре, пять или двадцать флагов? Трудно представить себе код реализации, который будет обрабатывать все эти комбинации сравнений.

Но здесь сильно поможет преобразование логических значений в числа (очевидно, 0 или 1):

```
function onlyOne() {  
    var sum = 0;  
    for (var i=0; i < arguments.length; i++) {  
        // Ложные значения пропускаются. Эквивалентно  
        // их интерпретации как 0, но избегает NaN.  
        if (arguments[i]) {  
            sum += arguments[i];  
        }  
    }  
    return sum == 1;  
}  
  
var a = true;  
var b = false;  
  
onlyOne( b, a );           // true  
onlyOne( b, a, b, b, b ); // true  
  
onlyOne( b, b );           // false  
onlyOne( b, a, b, b, b, a ); // false
```



Конечно, вместо цикла `for` в `onlyOne(..)` можно было воспользоваться более компактной функцией ES5 `reduce(..)`, но я не хотел скрывать концепции за подробностями реализации.

Здесь значения 1 представляют `true`/истинность; полученные значения суммируются в числовом виде. Чтобы это происходило, `sum += arguments[i]` использует неявное преобразование. Если одно (и только одно!) значение в списке аргументов истинно, то числовая сумма будет равна 1. Если сумма не равна 1, значит, нужное условие не выполняется.

Конечно, то же самое можно сделать с явным преобразованием:

```
function onlyOne() {  
    var sum = 0;  
    for (var i=0; i < arguments.length; i++) {
```

```
    sum += Number( !!arguments[i] );
}
return sum === 1;
}
```

Сначала конструкция `!!arguments[i]` обеспечивает преобразование значения в `true` или `false`. Это позволяет передавать при вызове нелогические значения, например `onlyOne("42", 0)`, и функция все равно будет работать как ожидалось (без `!!` будет выполнена конкатенация строк, и логика будет нарушена).

Убедившись в том, что значение имеет тип `boolean`, мы выполняем еще одно неявное преобразование с `Number(..)`, чтобы значение было равно `0` или `1`.

Можно ли сказать, что форма с неявным преобразованием «лучше»? Она избегает ловушки с `NaN`, упоминаемой в комментариях в коде. Но в итоге все зависит от ваших потребностей. Лично я считаю, что первая версия, основанная на неявном преобразовании, более элегантна (если вы не будете передавать `undefined` или `NaN`), а явная версия получается более длинной без какой-либо пользы.

Но как практически во всех случаях, которые здесь рассматриваются, это дело вкуса.



Независимо от выбранного подхода (явного или неявного) вы всегда можете легко создать вариации для другого количества условий `onlyTwo(..)`, `onlyFive(..)` и т. д., просто изменив последнее сравнение с `1` на `2`, `5` и т. д. Это гораздо проще, чем добавлять новые выражения `&&` и `||`. Преобразования типа обычно очень полезны в подобных ситуациях.

Неявные преобразования: * --> Boolean

А теперь обратимся к неявному преобразованию к логическим значениям. Пожалуй, это самый распространенный случай и, безусловно, самый потенциально опасный.

Помните: неявное преобразование вступает в силу тогда, когда вы используете значение в контексте, в котором это значение должно быть преобразовано. Для числовых и строковых операций нетрудно понять, как могут выполняться такие преобразования.

Но какие операции могут потребовать (неявного) преобразования к `boolean`?

1. Условие в команде `if (..)`.
2. Условие (вторая часть) в заголовке `for (.. ; .. ; ..)`.
3. Условие в циклах `while (..)` и `do..while(..)`.
4. Условие (первая часть) в тернарных выражениях `? ..`.
5. Левый operand (который служит проверяемым условием — см. ниже!) для операторов `||` («логическое OR») или `&&` («логическое AND»).

Любое значение, используемое в этих контекстах, которое не относится к типу `boolean`, будет неявно преобразовано к `boolean` по правилам абстрактной операции `ToBoolean` (см. ранее).

Несколько примеров:

```
var a = 42;
var b = "abc";
var c;
var d = null;

if (a) {
    console.log( "да" );           // да
}

while (c) {
    console.log( "неа, никогда не запустится" );
}

c = d ? a : b;
```

```
c;                                // "abc"  
  
if ((a && d) || c) {  
    console.log( "да" );           // да  
}
```

Во всех перечисленных контекстах нелогические значения неявно преобразуются в свои логические эквиваленты для принятия решений по условиям.

Операторы `||` и `&&`

Скорее всего, вы уже видели операторы `||` («логическое OR») и `&&` («логическое AND») практически во всех языках, на которых вы работали. Будет естественно предположить, что в JavaScript они работают, по сути, так же, как и в других языках.

Здесь существует очень малоизвестный, но при этом очень важный нюанс. Я бы даже сказал, что эти операторы не следовало называть «логическими операторами __», так как это название неточно описывает то, что они делают. Если бы мне предложили дать более точное (хотя и менее удобное) название, я бы назвал их «операторами выбора», а точнее «операторами выбора операнда».

Почему? Потому что в JavaScript они не дают логическое значение (то есть `boolean`), как в других языках.

Тогда какое значение *выдают* эти операторы? Значение одного (и только одного) из двух операндов. Иначе говоря, они выбирают значение одного из двух своих операндов.

Цитируя спецификацию ES5 из раздела 11.11:

«Значение, произведенное оператором `&&` или `||`, не обязательно относится к типу `Boolean`. Результатирующее значение всегда будет значением одного из двух выражений-операндов».

Продемонстрируем сказанное на примере:

```
var a = 42;  
var b = "abc";  
var c = null;  
  
a || b;      // 42  
a && b;      // "abc"  
  
c || b;      // "abc"  
c && b;      // null
```

Погодите, что?! Подумайте. В таких языках, как C и PHP, эти выражения дают результат `true` или `false`, но в JS (а также в Python и Ruby, если уж на то пошло) результат берется из самих значений. Операторы `||` и `&&` выполняют логическую проверку первого операнда (`a` или `c`). Если operand не относится к типу `boolean` (как в данном случае), происходит нормальное преобразование к `ToBoolean` для выполнения проверки.

Что касается оператора `||`, если условие истинно, то результатом выражения `||` становится значение *первого операнда* (`a` или `c`). Если же условие ложно, то результатом выражения `||` становится значение *второго операнда* (`b`).

И наоборот, для оператора `&&`, если условие истинно, то результатом выражения `&&` становится значение *второго операнда* (`b`). Если же условие ложно, то результатом выражения `&&` становится значение *первого операнда* (`a` или `c`).

Результат `||` или `&&` всегда совпадает со значением одного из operandов, а *не* с результатом проверки условия (возможно, преобразованным). В `c && b` значение `c` равно `null`, а следовательно, является ложным. Но к `null` приводит само выражение `&&` (значение `c`), а не преобразованное значение `false`, использованное при проверке.

Теперь вы видите, как операторы выполняют «выбор операнда»?

На эти операторы можно взглянуть иначе:

```
a || b;  
// приблизительно эквивалентно:  
a ? a : b;  
  
a && b;  
// приблизительно эквивалентно:  
a ? b : a;
```



Я назвал `a || b` «приблизительно эквивалентным» `a ? a : b`, потому что результат идентичен, но здесь существует тонкое отличие. В `a ? a : b`, если бы `a` было более сложным выражением (например, выражением с побочными эффектами — вызовом функции и т. д.), то выражение `a` могло бы быть вычислено дважды (если первое вычисление дало истинный результат). С другой стороны, для `a || b` выражение `a` будет вычислено только один раз, и это значение будет использовано как для проверки, так и в качестве результирующего значения (если подходит). Этот же нюанс существует в выражениях `a && b` и `a ? b : a`.

У этого поведения есть чрезвычайно распространенное и полезное применение. Возможно, вы уже использовали его ранее, но не понимали в полной мере:

```
function foo(a,b) {  
  a = a || "hello";  
  b = b || "world";  
  
  console.log( a + " " + b );  
}  
  
foo();           // "hello world"  
foo( "yeah", "yeah!" ); // "yeah yeah!"
```

Идиома `a = a || "hello"` (иногда называемая JavaScript-версией «оператором объединения с null») проверяет `a`, и при отсутствии значения (или если это нежелательное ложное значение) предоставляет резервное значение по умолчанию ("hello").

Но будьте внимательны!

```
foo( "That's it!", "" ); // "That's it! world" <-- ОЙ!
```

Видите проблему? "" как второй аргумент является ложным значением (см. «*ToBoolean*»), поэтому проверка `b = b || "world"` не проходит, и используется значение по умолчанию "world" — хотя, наверное, разработчик хотел, чтобы `b` было присвоено явно переданное значение "".

Идиома `||` встречается очень часто, она очень полезна, но ее следует использовать только в тех случаях, когда *все ложные значения* должны пропускаться. Иначе проверку условия в данном случае нужно сформулировать более явно; скорее всего, лучше воспользоваться тернарным оператором `? :`.

Идиома *присваивания со значением по умолчанию* настолько распространена (и удобна), что даже самые яростные противники преобразования типов JavaScript часто используют ее в своем коде.

А как насчет `&&`?

Существует другая идиома, которая значительно реже используется при «ручном» программировании, но часто используется минификаторами JS. Оператор `&&` «выбирает» второй operand в том и только в том случае, если первый operand дает истинное значение; этот прием иногда называется «защитным оператором» (см. также раздел «Ускоренная обработка» главы 5), первое выражение «проверяет» второе:

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

var a = 42;

a && foo(); // 42
```

`foo()` вызывается только в том случае, если `a` истинно. Если проверка не проходит, обработка выражения `a && foo()` просто прекращается, а функция `foo()` не вызывается.

Еще раз: эта конструкция редко встречается в коде, написанном людьми. Обычно вместо этого разработчики используют `if (a) { foo(); }`. Однако минификаторы JS выбирают `a && foo()`, потому что эта конструкция намного короче. Итак, если вам когда-либо приходилось расшифровывать такой код, вы знаете, что он делает и почему.

Итак, у `||` и `&&` в запасе есть несколько полезных трюков, если только вы не побоитесь задействовать неявное преобразование.



Идиомы `a = b || "something"` и `a && b()` полагаются на поведение ускоренной обработки, которое будет более подробно рассмотрено в разделе «Ускоренная обработка» главы 5.

Тот факт, что эти операторы не дают результат `true` или `false`, может вас немного встревожить. Возможно, вы думаете, как работали все ваши команды `if` и циклы `for`, если в них входили составные логические выражения вида `a && (b || c)`.

Не беспокойтесь! Ничего страшного не произошло. С вашим кодом (скорее всего) все нормально. Просто вы никогда не осознавали, что *после* вычисления составного выражения происходит неявное преобразование в `boolean`.

Пример:

```
var a = 42;
var b = null;
var c = "foo";

if (a && (b || c)) {
  console.log( "yep" );
}
```

Этот код всегда работал именно так, как вы ожидали, за исключением одного нюанса. Выражение `&& (b || c)` в действительности дает результат "foo", а не `true`. Итак, команда `if` затем инициирует преобразование значения "foo" в `boolean`, что, конечно, дает `true`.

Видите? Причин для паники нет. Вероятно, ваш код безопасен. Но теперь вы больше знаете о том, как он выполняет свою работу.

А теперь вы понимаете, что в этом коде используется неявное преобразование. Если вы все еще относитесь к лагерю противников (неявных) преобразований, придется вернуться и выполнить все проверки в явном виде:

```
if (!!a && (!!b || !!c)) {  
    console.log( "уэр" );  
}
```

Удачи! Шучу!

Преобразование символических имен

До настоящего момента между явными и неявными преобразованиями почти не было никаких ощутимых различий, все сводилось только к удобочитаемости кода.

Однако символические имена ES6 создают в системе преобразований скрытую ловушку, о которой тоже нужно поговорить. По причинам, которые выходят далеко за пределы материала, обсуждаемого в книге, явное преобразование символического имени в строку разрешено, но такое же неявное преобразование запрещено, а при попытке его выполнения происходит ошибка.

Пример:

```
var s1 = Symbol( "cool" );  
String( s1 );      // "Symbol(cool)"
```

```
var s2 = Symbol( "not cool" );
s2 + "";      // TypeError
```

Значения `symbol` вообще невозможно преобразовать в `number` (в любом случае происходит ошибка), но, как ни странно, они могут явно и неявно преобразовываться к `boolean` (результат всегда равен `true`).

Последовательное поведение всегда проще изучать, а с исключениями никогда не хочется иметь дела, но вы должны быть осторожны с новыми значениями `symbol`, появившимися в ES6, и способом их преобразования.

Хорошие новости: вероятно, необходимость в преобразовании значений `symbol` будет возникать крайне редко. Скорее всего, стандартный способ их использования (см. главу 3) не потребует их преобразования.

Равенство строгое и нестрогое

Нестрогое равенство проверяется оператором `==`, а строгое — оператором `===`. Оба оператора используются для сравнения двух значений на «равенство», но выбор формы (строгое/нестрогое) приводит к очень важным различиям в поведении, особенно в том, как принимается решение о равенстве.

По поводу этих двух операторов существует распространенное заблуждение: `==` проверяет на равенство значения, а `===` проверяет на равенство как значения, так и типы». Звучит разумно, но неточно. В бесчисленных авторитетных книгах и блогах, посвященных JavaScript, говорится именно это, но, к сожалению, все они *ошибаются*.

Правильное описание выглядит так: `==` допускает преобразование типа при проверке равенства, а `===` запрещает преобразование типа».

Быстродействие проверки равенства

Остановитесь и подумайте, чем первое (неточное) объяснение отличается от второго (точного).

В первом объяснении кажется очевидным, что оператор `==` выполняет *больше работы*, чем `==`, потому что он *также* должен проверить тип.

Во втором объяснении оператор `==` выполняет больше работы, потому что при различных типах ему приходится проходить через преобразование типа.

Не попадайтесь в ловушку, в которую попадают многие. Не думайте, что это хоть как-то отразится на быстродействии программы, а `==` будет сколько-нибудь ощутимо медленнее `==`. Хотя преобразование занимает какое-то время, оно занимает считанные микросекунды (да, миллионные доли секунды).

Если вы сравниваете два значения одного типа, `==` и `==` используют идентичный алгоритм, поэтому если не считать мелких различий в реализации движка, они должны выполнять одну и ту же работу.

Если вы сравниваете два значения разных типов, быстродействие не является важным фактором. Вы должны спрашивать себя о другом: если я сравниваю два значения, хочу ли я, чтобы выполнялось преобразование типов или нет?

Если преобразование вам нужно, используйте нестрогое равенство `==`, а если преобразование нежелательно, то используйте строгое равенство `==`.



Оба оператора, `==` и `==`, проверяют типы своих операндов. Различие в том, как они реагируют на несовпадение типов.

Абстрактная проверка равенства

Поведение оператора `==` определяется в разделе 11.9.3 спецификации ES5 («Алгоритм абстрактной проверки равенства»). Здесь приведен подробный, но простой алгоритм, с явным перечислением всех возможных комбинаций типов и способов преобразований типов (при необходимости), которые должны применяться в каждой комбинации.



Когда кто-то осуждает (неявное) преобразование типа как слишком сложное и содержащее слишком много дефектов для полезного практического применения, он осуждает именно правила «абстрактной проверки равенства». Обычно говорят, что этот механизм слишком сложен и противоестествен для практического изучения и использования, и что он скорее создает в JS-программах ошибки, чем упрощает чтение кода.

Я считаю, что это ошибочное предположение — ведь вы, читатели, являетесь компетентными разработчиками, которые пишут алгоритмы, то есть код (а также читают и разбираются в нем), целыми днями напролет. По этой причине я постараюсь объяснить «абстрактную проверку равенства» простыми словами. Однако я также рекомендую прочитать раздел 11.9.3 спецификации ES5. Думаю, вас удивит, насколько там все логично.

По сути, первый раздел (11.9.3.1) утверждает, что, если два сравниваемых значения относятся к одному типу, они сравниваются простым и естественным способом. Например, 42 равно только 42, а строка "abc" равна только "abc".

Несколько второстепенных исключений, о которых следует помнить:

- Значение `NaN` никогда не равно само себе (см. главу 2).
- `+0` и `-0` равны друг другу (см. главу 2).

Последняя секция в разделе 11.9.3.1 посвящена нестрогой проверке равенства `==` с объектами (включая функции и массивы). Два таких значения равны *только* в том случае, если оба они ссылаются в точности *на одно значение*. Никакое преобразование типа при этом не выполняется.



Строгая проверка равенства `==` определяется идентично 11.9.3.1, включая положение о двух объектных значениях. Этот факт очень малоизвестен, но `==` и `==` при сравнении двух объектов ведут себя полностью идентично!

Оставшаяся часть алгоритма в 11.9.3 указывает, что нестрогое равенство `==` может использоваться для сравнения двух разнотипных значений, одно или оба из которых потребуют неявного преобразования. В результате преобразования оба значения приводятся к одному типу, после чего они могут напрямую сравниваться на равенство по простой идентичности значений.



Операция нестрогой проверки неравенства `!=` определяется в точности так, как следовало ожидать; по сути операция `==` выполняется в полной мере, с последующим вычислением отрицания результата. То же относится к операции строгой проверки неравенства `!==`.

Сравнение: строки и числа

Для демонстрации преобразования `==` сначала создадим примеры строки и числа, что было сделано ранее в этой главе:

```
var a = 42;
var b = "42";

a === b;      // false
a == b;       // true
```

Как и следовало ожидать, проверка `a === b` завершается неудачей, поскольку преобразование не разрешено, а значения `42` и `"42"` различны.

Однако во втором сравнении `a == b` используется нестрогое равенство; это означает, что если типы окажутся различными, алгоритм сравнения выполнит неявное преобразование одного или обоих значений.

Но какое именно преобразование здесь выполняется? Станет ли значение `a`, то есть `42`, строкой, или же значение `b` `"42"` станет числом? В спецификации ES5 в разделах 11.9.3.4–5 говорится:

1. Если `Type(x)` относится к типу `Number`, а `Type(y)` относится к типу `String`, вернуть результат сравнения `x == ToNumber(y)`.
2. Если `Type(x)` относится к типу `String`, а `Type(y)` относится к типу `Number`, вернуть результат сравнения `ToNumber(x) == y`.



В спецификации используются формальные имена типов `Number` и `String`, тогда как в книге для примитивных типов обычно используются обозначения `number` и `string`. Не путайте регистр символа `Number` в спецификации со встроенной функцией `Number()`. Для наших целей регистр символов в имени типа роли не играет — они означают одно и то же.

Спецификация говорит, что значение `"42"` преобразуется в число для сравнения. О том, *как именно* выполняется преобразование, уже было рассказано ранее, а конкретно при описании абстрактной операции `ToNumber`. В этом случае вполне очевидно, что полученные два значения `42` равны.

Сравнение: что угодно с логическими значениями

Одна из самых опасных ловушек при неявном преобразовании типа `==` встречается при попытке прямого сравнения значения с `true` или `false`.

Пример:

```
var a = "42";
var b = true;

a == b; // false
```

Погодите, что здесь происходит? Мы знаем, что "42" является истинным значением (см. ранее в этой главе). Как же получается, что сравнение его с `true` оператором нестрогого равенства `==` не дает `true`?

Причина проста и одновременно обманчиво хитроумна. Ее легко понять неправильно, многие разработчики JS не прикладывают должных усилий, чтобы полностью разобраться в ней.

Еще раз процитируем спецификацию, разделы 11.9.3.6–7:

1. Если `type(x)` относится к типу `Boolean`, вернуть результат сравнения `ToNumber(x) == y`.
2. Если `type(y)` относится к типу `Boolean`, вернуть результат сравнения `x == ToNumber(y)`.

Посмотрим, что здесь. Первый шаг:

```
var x = true;
var y = "42";

x == y; // false
```

`Type(x)` действительно относится к типу `Boolean`, поэтому выполняется операция `ToNumber(x)`, которая преобразует `true` в `1`. Теперь вычисляется условие `1 == "42"`. Типы все равно различны, поэтому (фактически рекурсивно) алгоритм повторяется; как и в предыдущем случае, "42" преобразуется в `42`, а условие `1 == 42` очевидно ложно.

Если поменять операнды местами, результат останется прежним:

```
var x = "42";
var y = false;

x == y; // false
```

На этот раз `Type(y)` имеет тип `Boolean`, так что `ToNumber(y)` дает `0`. Условие `"42" == 0` рекурсивно превращается в `42 == 0`, что, разумеется, ложно.

Другими словами, значение `"42"` ни `== true`, ни `== false`. На первый взгляд это утверждение кажется совершенно немыслимым. Как значение может быть ни истинным, ни ложным?

Но в этом и заключается проблема! Вы задаете совершенно не тот вопрос. Хотя на самом деле это не ваша вина, это вас обманывает мозг.

Значение `"42"` действительно истинно, но конструкция `"42" == true` вообще не выполняет проверку `boolean`/преобразование, что бы там ни говорил ваш мозг. `"42" не преобразуется в boolean (true);` вместо этого `true` преобразуется в `1`, а затем `"42"` преобразуется в `42`.

Нравится вам это или нет, `ToBoolean` здесь вообще не используется, так что истинность или ложность `"42"` вообще не важна для операции `==!` Важно понимать, как алгоритм сравнения `==` ведет себя во всех разных комбинациях типов. Если с одной из сторон стоит логическое значение `boolean`, то оно всегда сначала преобразуется в число.

Если это кажется вам странным, вы не одиноки. Лично я рекомендую никогда, никогда, ни при каких обстоятельствах не использовать `== true` или `== false`. Никогда.

Но помните, что я здесь говорю только о `==`. Конструкции `== true` и `== false` не допускают преобразование типа, поэтому они защищены от скрытого преобразования `ToNumber`.

Пример:

```
var a = "42";  
  
// плохо (проверка не проходит!):  
if (a == true) {  
    // ..  
}  
  
// тоже плохо (проверка не проходит!):  
if (a === true) {  
    // ..  
}  
  
// достаточно хорошо (неявное преобразование):  
if (a) {  
    // ..  
}  
  
// лучше (явное преобразование):  
if (!!a) {  
    // ..  
}  
  
// тоже хорошо (явное преобразование):  
if (Boolean( a )) {  
    // ..  
}
```

Если вы будете избегать `== true` или `== false` (нестрогое равенство с `boolean`) в своем коде, вам никогда не придется беспокоиться об этой ловушке, связанной с истинностью/ложностью.

Сравнение: `null` с `undefined`

Другой пример неявного преобразования встречается при использовании нестрогой проверки равенства `==` между значениями `null` и `undefined`. Снова процитирую спецификацию ES5, разделы 11.9.3.2–3:

1. Если `x` содержит `null`, а `y` содержит `undefined`, вернуть `true`.
2. Если `x` содержит `undefined`, а `y` содержит `null`, вернуть `true`.

`Null` и `undefined` при сравнении нестрогим оператором `==` равны друг другу (то есть преобразуются друг к другу), и никаким другим значениям во всем языке.

Для нас это означает то, что `null` и `undefined` могут рассматриваться как неразличимые для целей сравнения, если вы используете нестрогий оператор проверки равенства `==`, разрешающий их взаимное неявное преобразование:

```
var a = null;
var b;

a == b;      // true
a == null;   // true
b == null;   // true

a == false; // false
b == false; // false
a == "";    // false
b == "";    // false
a == 0;     // false
b == 0;     // false
```

Преобразование между `null` и `undefined` безопасно и предсказуемо, и никакие другие значения не могут дать ложные положительные срабатывания при такой проверке. Я рекомендую использовать это преобразование, чтобы `null` и `undefined` не различались в программе и интерпретировались как одно значение.

Пример:

```
var a = doSomething();

if (a == null) {
  // ..
}
```

Проверка `a == null` проходит только в том случае, если `doSomething()` вернет `null` или `undefined`, и не пройдет при любом другом значении (включая `0`, `false` и `""`).

Явная форма этой проверки, которая запрещает любые подобные преобразования типов, выглядит (на мой взгляд) намного уродливее и, возможно, работает чуть менее эффективно!

```
var a = doSomething();

if (a === undefined || a === null) {
  // ..
}
```

Я считаю, что форма `a == null` — еще один пример ситуации, в которой неявное преобразование упрощает чтение кода, но делает это надежно и безопасно.

Сравнение: объекты и необъекты

Если объект/функция/массив сравнивается с простым скалярным примитивом (`string`, `number` или `boolean`), в спецификации ES5 говорится следующее (раздел 11.9.3.8–9):

1. Если `Type(x)` относится к типу `String` или `Number`, а `Type(y)` относится к типу `Object`, вернуть результат сравнения `x == ToPrimitive(y)`.
2. Если `Type(x)` относится к типу `Object`, а `Type(y)` относится к типу `String` или `Number`, вернуть результат сравнения `ToPrimitive(x) == y`.



Возможно, вы заметили, что в этих разделах спецификации упоминаются только `String` и `Number`, но не `Boolean`. Дело в том, что, как упоминалось выше, разделы 11.9.3.6–7 гарантируют, что любой операнд `Boolean` был сначала представлен в виде `Number`.

Пример:

```
var a = 42;
var b = [ 42 ];
a == b; // true
```

Для значения [42] вызывается абстрактная операция `ToPrimitive` (см. «Абстрактные операции»), которая дает результат "42". С этого момента остается простое условие "42" == 42, которое, как мы уже выяснили, превращается в 42 == 42, так что `a` и `b` равны с точностью до преобразования типа.



Как и следовало ожидать, все особенности абстрактной операции `ToPrimitive`, которые рассматривались ранее в этой главе (`(toString(), valueOf())`, применимы и в данном случае. Это может быть весьма полезно, если у вас имеется сложная структура данных и вы хотите определить для нее специализированный метод `valueOf()`, который должен будет предоставлять простое значение для целей проверки равенства.

В главе 3 рассматривалась «распаковка» объектной обертки вокруг примитивного значения (как в `new String("abc")`, например), в результате чего возвращается нижележащее примитивное значение ("abc"). Это поведение связано с преобразованием `ToPrimitive` в алгоритме `==`:

```
var a = "abc";
var b = Object( a ); // то же, что `new String( a )`  

a === b; // false  

a == b; // true
```

`a == b` дает `true`, потому что `b` преобразуется (или «распаковывается») операцией `ToPrimitive` в базовое простое скалярное примитивное значение "abc", которое совпадает со значением из `a`.

Есть некоторые значения, для которых это не так из-за других переопределяющих правил в алгоритме `==`. Пример:

```
var a = null;
var b = Object( a );      // то же, что `Object()`
a == b;                  // false

var c = undefined;
var d = Object( c );      // то же, что `Object()`
c == d;                  // false

var e = NaN;
var f = Object( e );      // то же, что `new Number( e )`
e == f;                  // false
```

Значения `null` и `undefined` не могут упаковываться (у них нет эквивалентной объектной обертки), так что `Object(null)` принципиально не отличается от `Object()`: оба вызова создают обычный объект.

`NaN` можно упаковать в эквивалентную объектную обертку `Number`, но когда `==` вызывает распаковку, сравнение `NaN == NaN` не проходит, потому что значение `NaN` никогда не равно самому себе (см. главу 2).

Особые случаи

Итак, мы тщательно разобрались в том, как работает неявное преобразование при нестрогой проверке равенства `==` (как разумные, так и странные аспекты). А теперь попробуем найти самые худшие, самые сумасшедшие особые случаи. Вы будете знать, от чего стоит держаться подальше, чтобы не столкнуться с ошибками преобразования.

Для начала посмотрим, как изменение встроенных прототипов позволит избежать невероятных результатов:

```
Number.prototype.valueOf = function() {
  return 3;
};

new Number( 2 ) == 3;    // true
```



Для проверки `2 == 3` эта ловушка не сработает, потому что ни для 2, ни для 3 не будет вызван встроенный метод `Number.prototype.valueOf()` — оба значения уже являются встроенными числовыми значениями, которые могут сравниваться напрямую. Тем не менее вызов `new Number(2)` должен пройти через преобразование `ToPrimitive`, а следовательно, потребует вызова `valueOf()`.

Скверно, да? Еще бы. Никогда так не делайте. Сама теоретическая возможность иногда используется для критики преобразований типов и `==`. Тем не менее эта критика направлена не по адресу. Язык JavaScript не плох из-за того, что он позволяет делать подобные вещи; это разработчик плох, если он их делает. Не впадайте в заблуждение «язык программирования должен защищать меня от меня самого».

Теперь рассмотрим другой нетривиальный пример, который поднимает скверну предыдущего примера на новый уровень:

```
if (a == 2 && a == 3) {  
    // ..  
}
```

Возможно, вы думаете, что это невозможно, потому что `a` не может быть равно 2 и 3 *одновременно*. Но выражение «одновременно» в данном случае неточно, так как первое выражение `a == 2` вычисляется строго ранее `a == 3`.

Итак, что произойдет, если у `a.valueOf()` будут побочные эффекты при каждом вызове, так что в первый раз он вернет 2, а во втором 3? Проще простого:

```
var i = 2;  
  
Number.prototype.valueOf = function() {  
    return i++;  
};  
  
var a = new Number( 42 );
```

```
if (a == 2 && a == 3) {  
  console.log( "Да, это случилось." );  
}
```

Еще раз подчеркну: все это скверные трюки. Не используйте их. Также их не стоит использовать как претензии к преобразованию типов. Потенциальные злоупотребления механизмом не могут считаться достаточным доказательством для его обвинения. Просто держитесь подальше от этого безумия и старайтесь правильно использовать преобразование типов.

Сравнения ложных значений

Самые распространенные претензии к неявным преобразованиям при сравнениях `==` происходят от странностей поведения ложных значений при сравнении их друг с другом.

Для наглядности рассмотрим список особых случаев при сравнениях ложных значений. Вы сами увидите, какие варианты разумны, а какие создают проблемы:

```
"0" == null;          // false  
"0" == undefined;    // false  
"0" == false;         // true - ОЙ-ОЙ!  
"0" == NaN;          // false  
"0" == 0;             // true  
"0" == "";            // false  
  
false == null;        // false  
false == undefined;   // false  
false == NaN;         // false  
false == 0;            // true - ОЙ-ОЙ!  
false == "";           // true - ОЙ-ОЙ!  
false == [];           // true - ОЙ-ОЙ!  
false == {};           // false  
  
"" == null;           // false  
"" == undefined;     // false  
"" == NaN;            // false  
"" == 0;               // true - ОЙ-ОЙ!
```

```
"" == [];           // true - ОЙ-ОЙ!
"" == {};           // false

0 == null;          // false
0 == undefined;     // false
0 == NaN;           // false
0 == [];           // true - ОЙ-ОЙ!
0 == {};           // false
```

В списке 24 сравнения, 17 выглядят вполне разумно и предсказуемо. Например, мы знаем, что `""` и `NaN` несравнимы; и действительно, они не преобразуются в значения, равные для нестрогого сравнения, тогда как `0` и `0` сравнимы в достаточной мере и преобразуются в значения, равные для нестрогого сравнения.

Семь сравнений снабжены пометкой «ОЙ-ОЙ!», потому что они как ложные положительные срабатывания с много большей вероятностью вызовут проблемы, о которые вы можете споткнуться. `""` и `0` определенно являются разными значениями, и вам вряд ли когда-нибудь захочется проверять их на равенство, так что взаимное преобразование чревато проблемами. Обратите внимание: в списке нет ни одного ложного отрицательного срабатывания.

Самые странные случаи

Впрочем, на этом мы не остановимся, а продолжим поиск еще более опасных преобразований:

```
[] == ![];           // true
```

Так-так! Безумие выходит на новый уровень? Возможно, ваш мозг обманывает вас, подсказывая, что истинное значение сравнивается с ложным, так что результат `true` покажется удивительным, ведь значение никогда не может быть истинным и ложным одновременно!

Но на самом деле здесь происходит совсем другое. Разобьем происходящее на части. Что вы знаете об унарном операторе `!`?

Он явно преобразуется в `boolean` по правилам `ToBoolean` (а также переключает состояние битов). Итак, еще перед обработкой условие `[] == ![]` уже преобразуется в `[] == false`. Вы уже видели эту форму в приведенном выше списке (`false == []`), так что в удивительном результате на самом деле нет ничего нового.

Как насчет других особых случаев?

```
2 == [2];      // true
"" == [null];  // true
```

Как упоминалось ранее в обсуждении `ToNumber`, значения `[2]` и `[null]` в правой части проходят преобразование `ToPrimitive`, чтобы их было проще сравнивать с простыми примитивами (`2` и `""` соответственно) в левой части. Так как `valueOf()` для массива просто возвращает сам массив, преобразование переходит к преобразованию массива в строку.

`[2]` превращается в `"2"`, после чего `ToNumber` преобразует это значение в `2` для значения в правой части при первом сравнении. `[null]` просто превращается в `""`.

Условия `2 == 2` и `"" == ""` полностью понятны.

Если вы все еще инстинктивно противитесь этим результатам, ваше раздражение на самом деле направлено не на преобразование, как можно было бы подумать. На самом деле оно направлено на поведение по умолчанию `ToPrimitive` для массивов, которые преобразуются в значение `string`. Скорее вы бы предпочли, чтобы вызов `[2].toString()` не возвращал `"2"` или чтобы вызов `[null].toString()` не возвращал `""`.

Но что именно *должны* возвращать эти преобразования? Я не могу придумать никакого другого разумного преобразования `[2]` в `string`, кроме разве что `"2"`, но это может выглядеть очень странно в других контекстах!

Можно справедливо заметить, что поскольку `String(null)` превращается в `"null"`, то `String([null])` тоже должно превращаться

ся в `"null"`. Это вполне разумное допущение. В общем, мы добрались до настоящей причины всех бед.

Неявное преобразование само по себе не является злом. Даже явное преобразование `[null]` в строку дает `""`. Вопрос в том, насколько разумно для значений-массивов преобразовываться в эквивалент их содержимого и как именно это происходит. Итак, ваше раздражение следует направить на правила `String([...])`, потому что все странности возникают именно из-за них. Может, для массивов вообще не должно быть преобразования в строку? Но это будет иметь множество отрицательных последствий в других частях языка.

Еще один часто приводимый пример:

```
0 == "\n";           // true
```

Как обсуждалось ранее для пустых строк `""`, `"\n"` (или `" "`, или любая другая комбинация символов-пропусков) преобразуется с использованием `ToNumber`, а результат равен 0. К какому другому числовому значению должны приводиться пропуски? Вас беспокоит, что явное преобразование `Number(" ")` дает 0?

На самом деле есть еще только одно разумное числовое значение, к которому могли бы приводиться пустые строки или пропуски — это `NaN`. Но будет ли это лучше? Конечно, сравнение `" " == NaN` завершится неудачей, но решит ли это хоть одну проблему? Шансы на то, что в реальной JS-программе произойдет сбой из-за `0 == "\n"`, ничтожно малы, и такие особые случаи легко обходятся.

В любом языке у преобразований типа *всегда* существуют граничные случаи, преобразование типов ничем не отличается от них. Домысливание в некоторых особых ситуациях может создать проблемы (может, и заслуженные), но это нельзя считать убедительным аргументом против механизма преобразования типов в целом.

Мораль: почти любое странное преобразование между *обычными значениями*, с которыми вы можете столкнуться на практике (не считая намеренно хитроумных трюков с `valueOf()` или `toString()`, продемонстрированных выше), сведется к короткому списку опасных преобразований из 7 пунктов (с. 153).

Чтобы выделить их на фоне 24 «подозреваемых» для ловушек при преобразовании типов, рассмотрим другой список:

```
42 == "43";                                // false
"foo" == 42;                                 // false
"true" == true;                             // false

42 == "42";                                // true
"foo" == [ "foo" ];                          // true
```

В этих неложных, неособых случаях (а число сравнений, которые можно было бы включить в этот список, буквально бесконечно!) результаты преобразования типов полностью безопасны, разумны и объяснимы.

Немного здравого смысла

Окей, мы нашли некоторые странности при подробном изучении неявных преобразований. Неудивительно, что многие разработчики считают преобразование типов злом, от которого нужно держаться подальше.

Но давайте сделаем шаг назад и воспользуемся здравым смыслом.

Взглянем на цифры: у нас есть *список* из семи проблемных преобразований типов и *еще один* список (из 17 пунктов, но на самом деле бесконечный) преобразований абсолютно разумных и объяснимых.

Помните поговорку «Не выплескивайте ребенка вместе с водой»? Перед вами классический пример: не стоит полностью отказываться от преобразований типов (бесконечно большого списка

безопасных и полезных вариантов поведения) из-за списка, содержащего всего семь проблем.

Разумнее было бы спросить: «Как я могут использовать бесчисленные *хорошие* аспекты преобразования типов и при этом обойти несколько *плохих* аспектов?»

Снова рассмотрим список плохих преобразований:

```
"0" == false;           // true - ОЙ-ОЙ!
false == 0;             // true - ОЙ-ОЙ!
false == "";            // true - ОЙ-ОЙ!
false == [];            // true - ОЙ-ОЙ!
"" == 0;                // true - ОЙ-ОЙ!
"" == [];              // true - ОЙ-ОЙ!
0 == [];               // true - ОЙ-ОЙ!
```

Четыре из семи пунктов включают сравнение `== false`, которое, как было сказано выше, *никогда, никогда* не следует использовать. Это правило запоминается довольно легко.

Список сокращается до трех пунктов:

```
"" == 0;                // true - ОЙ-ОЙ!
"" == [];              // true - ОЙ-ОЙ!
0 == [];               // true - ОЙ-ОЙ!
```

Все это разумные преобразования, которые вы бы использовали в нормальной JS-программе? При каких условиях они могут вам встретиться?

Маловероятно, чтобы вам захотелось буквально использовать `== []` в логическом условии в вашей программе, разве что вы точно знаете, что вы делаете. Скорее будет использована проверка `== ""` или `== 0`:

```
function doSomething(a) {
  if (a == "") {
    // ..
  }
}
```

Если вы случайно вызовете `doSomething(0)` или `doSomething([])`, вас ждет неприятный сюрприз. Другой сценарий:

```
function doSomething(a,b) {  
  if (a == b) {  
    // ..  
  }  
}
```

И снова работоспособность программы будет нарушена, если вы используете вызов вида `doSomething("",0)` или `doSomething([], "")`.

Итак, хотя в отдельных ситуациях эти преобразования *могут* вас подвести и с ними необходима осторожность, скорее всего, они будут довольно редко встречаться в вашей кодовой базе.

Безопасное использование неявного преобразования

Самый важный совет, который я могу вам дать: проанализируйте свою программу и подумайте над тем, какие значения могут стоять в обеих частях сравнения `==`. Пара эвристических правил поможет вам избежать проблем с такими сравнениями:

- Если в одной из частей сравнения может находиться значение `true` или `false`, никогда, НИКОГДА не используйте `==`.
- Если в одной из частей сравнения может находиться значение `[]`, `""` или `0`, постарайтесь обойтись без использования `==`.

В этих сценариях практически всегда лучше использовать `==` вместо `==`, чтобы избежать нежелательных преобразований типов. Соблюдайте эти два простых правила, и вы обойдете почти все проблемы с преобразованием типов, которые могут вас подвести.

Если вы будете более явно и подробно выражать свои намерения в коде, это избавит вас от многих неприятностей.

Вопрос об использовании `==` или `===` на самом деле звучит так: хотите ли вы разрешить преобразования типов при сравнении или нет?

Такие преобразования могут принести пользу в очень многих ситуациях. Они позволяют более компактно выразить логику сравнения (например, с `null` и `undefined`).

В общем и целом, случаи, в которых неявное преобразование действительно опасно, относительно немногочисленны. Но в таких случаях для безопасности определенно следует использовать `===`.



Другое место, в котором преобразование типа гарантированно вас *не подведет*, — оператор `typeof`. `typeof` всегда возвращает одну из семи строк (см. главу 1), ни одна из которых не является пустой строкой `""`. Соответственно ни в каком случае проверка типа некоторого значения не пойдет на смарку из-за неявного преобразования. Условие `typeof x == "function"` на 100 % так же безопасно и надежно, как и условие `typeof x === "function"`. В спецификации указано, что в этих ситуациях применяются идентичные алгоритмы. Итак, не стоит бездумно использовать `==` повсюду просто потому, что ваши средства работы с кодом вам это приказывают, или (что самое худшее) потому, что в какой-то книге было написано, что вам не стоит над этим задумываться. Вы сами отвечаете за качество своего кода.

Неявное преобразование типов вредно и опасно? В отдельных случаях да, но в подавляющем большинстве случаев — нет.

Будьте ответственным и зрелым разработчиком. Научитесь пользоваться мощью преобразования типов (как явного, так и неявного) эффективно и безопасно и научите этому окружающих.

На рис. 4.1 изображена удобная таблица, созданная пользователем GitHub Алексом Дори (Alex Dorey) (@dorey на GitHub) для наглядного представления разнообразных сравнений.

Рис. 4.1. Проверка равенства в JavaScript¹¹ <https://github.com/dorey/JavaScript-Equality-Table>.

Абстрактное относительное сравнение

Хотя этой части неявных преобразований часто уделяется гораздо меньшее внимание, важно думать о том, что произойдет со сравнениями `a < b` (по аналогии с тем, как мы подробно анализировали `a == b`). Алгоритм «Абстрактное относительное сравнение» в ES5 (раздел 11.8.5) фактически делится на две части: что нужно делать, если в сравнении задействованы два строковых значения (вторая половина), и во всех остальных случаях (первая половина).



Алгоритм определен только для `a < b`. Таким образом, `a > b` обрабатывается как `b < a`.

Алгоритм сначала вызывает преобразование `ToPrimitive` для обоих значений. Если возвращаемый результат хотя бы одного вызова не является строкой, то оба значения преобразуются в числовые значения по правилам операции `ToNumber` и сравниваются в числовом виде.

Пример:

```
var a = [ 42 ];
var b = [ "43" ];

a < b; // true
b < a; // false
```



В данном случае действуют те же предупреждения для `-0` и `NaN`, как и для рассмотренного выше алгоритма `==`.

Но если при сравнении `<` оба значения являются строками, выполняется простое лексикографическое (естественное алфавитное) сравнение символов:

```
var a = [ "42" ];
var b = [ "043" ];

a < b; // false
```

а и b *не* преобразуются в числа, потому что оба значения становятся строками после преобразования `ToPrimitive` для двух массивов. Итак, "42" сравнивается (символ за символом) с "043", начиная с первых символов «4» и «0» соответственно. Так как "0" в лексикографическом отношении *меньше* "4", сравнение возвращает `false`.

То же поведение и рассуждения применяются в следующем примере:

```
var a = [ 4, 2 ];
var b = [ 0, 4, 3 ];

a < b; // false
```

Здесь a превращается в "4,2", а b превращается в "0,4,3". Лексикографическое сравнение этих двух строк выполняется так же, как в предыдущем фрагменте.

А как насчет такого примера:

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // ??
```

a < b тоже дает `false`, потому что a превращается в `[object Object]`, b тоже превращается в `[object Object]`; очевидно, a не будет лексикографически меньше b.

Но как ни странно:

```
var a = { b: 42 };
var b = { b: 43 };

a < b; // false
a == b; // false
```

```
a > b; // false
```

```
a <= b; // true
```

```
a >= b; // true
```

Почему `a == b` не равно `true`? Это одинаковые строковые значения (`"[object Object]"`), поэтому они должны быть равны, правильно? Нет. Вспомните предыдущее обсуждение относительно того, как `=` работает с объектными ссылками. Но тогда как `a <= b` и `a >= b` могут дать `true`, если все условия `a < b`, `a == b` и `a > b` дают `false`?

Потому что в спецификации сказано, что для `a <= b` сначала будет вычислено условие `b < a`, после чего результат будет инвертирован. Поскольку условие `b < a` *тоже* ложно, результат `a <= b` равен `true`.

Возможно, это сильно противоречит вашим прежним представлениям о принципах работы `<=`, скорее всего, вы рассматривали его как литерал «меньше либо равно». JS более точно интерпретирует `<=` как «не больше чем» (`!(a > b)`), что JS интерпретирует как `!(b < a)`. Более того, для объяснения `a >= b` условие сначала рассматривается в виде `b <= a`, после чего применяются те же рассуждения.

К сожалению, здесь не существует «строгого относительного сравнения», какое существует для равенства. Другими словами, с относительными сравнениями вида `a < b` невозможно предотвратить неявное преобразование, кроме как явно привести `a` и `b` к одному типу перед сравнением.

Воспользуемся рассуждениями из приведенного ранее сравнительного обсуждения `==` и `===`. Если преобразование типа полезно и относительно безопасно, как при сравнении `42 < "43"` — используйте его. С другой стороны, если вам нужны гарантии безопасности при относительных сравнениях, выполните явное преобразование значений перед использованием `<` (или другого аналогичного оператора):

```
var a = [ 42 ];
var b = "043";

a < b;           // false -- сравнение строк!
Number( a ) < Number( b ); // true -- сравнение чисел!
```

Итоги

В этой главе мы разбирались с тем, как в JavaScript выполняются преобразования типов. Все преобразования можно разделить на явные и неявные.

У преобразования типов дурная репутация, но на самом деле оно достаточно полезно во многих ситуациях. Очень важно, чтобы ответственный JS-разработчик выделил время на изучение всех тонкостей преобразований типов и решил для себя, какие их части помогут улучшить его код, а от каких следует держаться подальше.

Явным преобразованием называется конструкция, очевидным предназначением которой является преобразование значения из одного типа к другому. Основные преимущества явных преобразований — улучшение ясности и удобства сопровождения кода за счет сокращения путаницы.

Неявное преобразование скрыто; оно является побочным эффектом некоторой другой операции, по внешнему виду которой не очевидно, что произойдет преобразование типа. Хотя может показаться, что неявное преобразование является противоположностью явного, и по этой причине нежелательно (и многие в самом деле так думают), неявные преобразования также направлены на улучшение удобочитаемости кода.

Преобразования, особенно неявные, должны использоваться ответственно и сознательно. Вы должны понимать, почему вы пишете тот код, который вы пишете, и как он работает. Страйтесь написать такой код, который будет понятен другим разработчикам и по которому они смогут учиться.

5 Грамматика

Последняя серьезная тема, которую мы здесь рассмотрим, — это работа синтаксиса языка JavaScript (то есть грамматика JavaScript). Возможно, вы думаете, что уже знаете, как нужно писать на JS, однако в различных частях языка существует огромное количество нюансов, которые ведут к путанице и недопониманию. Мы разберемся с этими частями и постараемся прояснить ситуацию.



Возможно, термин «грамматика» менее знаком читателям, чем термин «синтаксис». Во многих отношениях эти термины похожи; они описывают *правила*, по которым работает язык. Между ними существуют тонкие различия, но в основном они не имеют значения для нашего обсуждения. Грамматика JavaScript представляет собой структурированный способ описания того, как элементы синтаксиса (операторы, ключевые слова и т. д.) объединяются в стройную, корректную программу. Другими словами, при рассмотрении синтаксиса без грамматики мы упустим многие важные подробности. По этой причине наша основная тема в этой главе точнее всего описывается термином «грамматика», хотя разработчики работают с базовым синтаксисом языка.

Команды и выражения

Разработчики довольно часто считают, что термины «команда» и «выражение» приблизительно эквивалентны. Но их необходимо различать, потому что в JS-программах это важно.

Чтобы обозначить различия, мы позаимствуем терминологию из области, которая должна быть вам знакома лучше: из нашего языка.

«Предложением» называется полная последовательность слов, выражающая некоторую мысль. Предложение состоит из одной или нескольких «фраз», связанных знаками препинания или союзами («и», «или» и т. д.). Сама фраза может состоять из меньших фраз. Некоторые фразы неполны и мало что дают сами по себе, тогда как другие фразы могут использоваться самостоятельно. Совокупность этих правил называется *грамматикой языка*.

Так же обстоит дело с грамматикой JavaScript. Команды являются аналогами предложений, выражения — аналогами фраз, а операторы — аналогами союзов/знаков препинания.

Каждое выражение в JS может быть вычислено с получением одного конкретного значения (результата). Пример:

```
var a = 3 * 6;  
var b = a;  
b;
```

В этом фрагменте `3 * 6` — выражение (при вычислении которого будет получен результат 18). Но `a` во второй строке тоже является выражением, как и `b` в третьей строке. Результатами вычисления выражений `a` и `b` являются значения, хранящиеся в этих значениях в настоящее время; они тоже равны 18.

Более того, каждая из трех строк является командой, содержащей выражения. `var a = 3 * 6` и `var b = a` называются «командами объявления», потому что они объявляют переменные (и могут присваивать им значения). Присваивания `a = 3 * 6` и `b = a` (без `var`) называются выражениями присваивания.

Третья строка содержит только выражение `b`, но она также сама является командой (хотя и не особо интересной). По этой причине она также называется «командой-выражением».

Завершающие значения команд

Малоизвестный факт: у каждой команды имеется завершающее значение (даже если это значение `undefined`).

Что бы вы сделали для того, чтобы просмотреть завершающее значение команды?

Самый очевидный ответ — ввести команду в консоли разработчика в браузере, потому что при выполнении консоль по умолчанию выводит завершающее значение последней выполненной команды.

Возьмем команду `var b = a`. Какое завершающее значение будет получено при выполнении этой команды?

Выражение присваивания `b = a` дает значение, которое было присвоено (18, как выше), но сама команда `var` дает результат `undefined`. Почему? Потому что команды `var` так определены в спецификации. Если ввести на консоли команду `var a = 42`, вы получите `undefined` вместо 42.



С технической точки зрения ситуация чуть сложнее. В спецификации ES5, раздел 12.2, алгоритм `VariableDeclaration` в действительности возвращает значение (строку с именем объявленной переменной, что, согласитесь, странно), но

это значение фактически поглощается (кроме использования в цикле `for..in`) алгоритмом `VariableStatement`, который требует пустого завершающего значения (то есть `undefined`).

Если вы уже экспериментировали с кодом в консоли (или в оболочке REPL среды JavaScript), вероятно, вы видели, что после многих команд выводится `undefined`, — и не понимали, откуда оно берется и что это такое. Проще говоря, консоль сообщает завершающее значение команды.

Но что бы ни выводилось в консоль, мы не можем перенести это значение в свою программу. Как получить завершающее значение в программе?

Это уже более сложная задача. Прежде чем объяснить, как это делается, сначала определим, для чего это может понадобиться.

Следует учитывать другие типы завершающих значений команд. Например, любой обычный блок `{ .. }` имеет завершающее значение — это завершающее значение последней выполненной в нем команды/выражения.

Пример:

```
var b;  
  
if (true) {  
  b = 4 + 38;  
}
```

Если ввести этот фрагмент в консоль/REPL, вероятно, вы увидите значение `42`, потому что `42` — завершающее значение блока `if`, которым становится завершающее значение последней команды-выражения `b = 4 + 38`.

Другими словами, завершающее значение блока аналогично значению, неявно возвращаемому последней командой блока.



На концептуальном уровне этот механизм близок к таким языкам, как CoffeeScript, в которых неявно возвращаемые значения функций равны значению последней выполненной команды в функции.

Но здесь возникает очевидная проблема. Такой код работать не будет:

```
var a, b;  
a = if (true) {  
    b = 4 + 38;  
};
```

Не существует простого синтаксиса/грамматики для получения возвращаемого значения команды и присваивания его другой переменной (по крайней мере, пока).

Что же делать?



Код приведен только для примера. Не делайте этого в реальном коде!

Для сохранения завершающего значения можно воспользоваться ненавистной функцией `eval(..)`:

```
var a, b;  
a = eval( "if (true) { b = 4 + 38; }" );  
a; // 42
```

Решение невероятно уродливо, но оно работает! И еще оно демонстрирует тот факт, что завершающие значения команд вполне реальны и их можно не только вывести в консоль, но и использовать в программах.

Для ES7 подано предложение с так называемыми «`do-выражениями`». Вот как они могут работать:

```
var a, b;  
  
a = do {  
    if (true) {  
        b = 4 + 38;  
    }  
};  
  
a; // 42
```

Выражение `do { ... }` выполняет блок (из одной или нескольких команд), а последнее завершающее значение команды внутри блока становится завершающим значением `do`-выражения, которое затем может быть присвоено `a`, как показано выше.

Общая идея заключается в том, чтобы интерпретировать команды как выражения (то есть включать их внутрь других команд) без необходимости заключать их во встроенные функциональные выражения и явно выполнять команду `return ...`.

А пока завершающие значения команд — в лучшем случае любопытный факт без особой практической ценности. Но вероятно, они станут намного более значимыми по мере развития JS. Будем надеяться, что выражения `do { ... }` избавят вас от соблазна использовать трюки с `eval(...)`.



Еще раз повторю свой совет: держитесь подальше от `eval(...)`. Серьезно.

Побочные эффекты выражений

Большинство выражений не имеет побочных эффектов. Пример:

```
var a = 2;  
var b = a + 3;
```

Выражение `a + 3` само по себе не имеет *побочного* эффекта (как, например, изменение `a`). У выражения есть результат (в данном случае `5`), и этот результат присваивается `b` в команде `b = a + 3`.

Самый распространенный пример выражения с (возможными) побочными эффектами — выражение с вызовом функции:

```
function foo() {  
    a = a + 1;  
}  
  
var a = 1;  
foo();      // результат: `undefined`, побочный эффект:  
           // изменение `a`
```

Впрочем, существуют и другие выражения с побочными эффектами. Пример:

```
var a = 42;  
var b = a++;
```

Выражение `a++` имеет два разных аспекта поведения. *Во-первых*, оно возвращает текущее значение `a`, равное `42` (которое затем присваивается `b`). Но *затем* оно изменяет значение самой переменной `a`, увеличивая ее на `1`:

```
var a = 42;  
var b = a++;  
  
a;  // 43  
b;  // 42
```

Многие разработчики ошибочно полагают, что `b` содержит значение `43`, как и `a`. Путаница происходит от неполного понимания побочных эффектов оператора `++`.

Оператор инкремента `++` и оператор декремента `--` являются унарными операторами (см. главу 4), которые могут использоваться либо в *постфиксном*, либо в *префиксном* варианте:

```
var a = 42;  
  
a++; // 42  
a; // 43  
  
++a; // 44  
a; // 44
```

Когда `++` используется в префиксном варианте (например, `++a`), его побочный эффект (увеличение `a`) происходит *до* возвращения значения из выражения, а не *после*, как в случае с `a++`.



Как вы думаете, `++a++` — это допустимый синтаксис? Если вы опробуете его в программе, то получите ошибку `ReferenceError`, но почему? Потому что операторам с побочными эффектами необходима ссылка на переменную, к которой будут применяться побочные эффекты. Для `++a++` сначала вычисляется часть `a++` (из-за приоритета операторов — см. ниже), которая дает значение `a` до инкремента. Но затем при попытке вычислить значение `++42` будет (если вы попробуете) выдана та же ошибка `ReferenceError`, так как `++` не может напрямую применять свой побочный эффект к значениям (например, `42`).

Иногда ошибочно считается, что побочный эффект `a++` можно инкапсулировать, заключив оператор в круглые скобки `()`:

```
var a = 42;  
var b = (a++);  
  
a; // 43  
b; // 42
```

К сожалению, круглые скобки `()` сами по себе не определяют новое выражение, которое будет вычислено *после побочного эффекта* выражения `a++`, как можно было надеяться. Даже если бы они определяли новое выражение, `a++` сначала вернет `42`. И, если только у вас нет другого выражения, которое заново вычисляет

а после побочного эффекта `++`, вы не получите 43 от этого выражения и `b` не будет присвоено значение 43.

Впрочем, существует одна возможность: оператор серии команд `,` (запятая). Этот оператор позволяет склеить несколько отдельных команд-выражений в одну команду:

```
var a = 42, b;  
b = ( a++, a );  
  
a; // 43  
b; // 43
```



Круглые скобки `(...)` вокруг `a++` здесь необходимы. Это связано с приоритетом оператора, который будет рассматриваться позже в этой главе.

Выражение `a++`, а означает, что второе подвыражение `a` будет вычисляться *после завершающих побочных эффектов* выражения `a++`; это означает, что оно вернет значение 43 для присваивания `b`.

Другой пример оператора с побочным эффектом — `delete`. Как было показано в главе 2, оператор `delete` используется для удаления свойства из объекта или элемента из массива. Но обычно он вызывается как отдельная команда:

```
var obj = {  
  a: 42  
};  
  
obj.a; // 42  
delete obj.a; // true  
obj.a; // undefined
```

Итоговое значение оператора `delete` равно `true`, если запрашиваемая операция действительна/допустима, или `false` в противном случае. Но у оператора есть и побочный эффект: удаление свойства (или элемента массива).



Что имеется в виду под «действительной/допустимой»? Для несуществующих свойств или существующих свойств с возможностью настройки оператор `delete` вернет `true`. В остальных случаях результатом будет `false` или ошибка.

Последним примером оператора с побочными эффектами, который может быть одновременно очевидным и неочевидным, является оператор присваивания `=`.

Пример:

```
var a;  
  
a = 42;      // 42  
a;          // 42
```

Может, и не очевидно, что `=` в `a = 42` является оператором с побочными эффектами для выражения. Но если проанализировать итоговое значение команды `a = 42`, это будет только что присвоенное значение `(42)`, так что присваивание того же значения `a` по сути является побочным эффектом.



Аналогичные рассуждения о побочных эффектах также применимы к операторам составного присваивания `+=`, `-=` и т. д. Например, `a = b += 2` сначала обрабатывается в форме `b += 2` (что эквивалентно `b = b + 2`), а результат присваивания = затем присваивается `a`.

Такое поведение, при котором выражение (или команда) присваивания приводит к присвоенному значению, в первую очередь полезно для сцепленных присваиваний, таких как:

```
var a, b, c;  
  
a = b = c = 42;
```

Здесь `c = 42` дает результат `42` (с побочным эффектом присваивания `42` переменной `c`), после чего для `b = 42` вычисляется результат `42` (с побочным эффектом присваивания `42` переменной `b`), и наконец, вычисляется результат `a = 42` (с побочным эффектом присваивания `42` переменной `a`).



Одна из распространенных ошибок, которые разработчики часто допускают со сцепленными присваиваниями, — конструкции вида `var a = b = 42`. Выглядит как одно и то же, но это не так. Если эта команда будет выполнена без отдельной команды `var b` (где-то в области видимости) для формального объявления `b`, то `var a = b = 42` не будет объявлять `b` напрямую. В зависимости от того, активен ли режим `strict`, либо произойдет ошибка, либо будет непреднамеренно создана глобальная переменная.

Другой сценарий, который следует рассмотреть:

```
function vowels(str) {
  var matches;

  if (str) {
    // извлечение гласных
    matches = str.match( /[aeiou]/g );

    if (matches) {
      return matches;
    }
  }
}

vowels( "Hello World" ); // [ "e", "o", "o" ]
```

Такое решение работает, и многие разработчики отдают ему предпочтение. Однако применение идиомы, использующее побочный эффект присваивания, позволяет упростить эту запись объединением двух команд `if` в одну:

```
function vowels(str) {  
  var matches;  
  
  // извлечение гласных  
  if (str && (matches = str.match( /[aeiou]/g ))) {  
    return matches;  
  }  
}  
  
vowels( "Hello World" ); // [ "e", "o", "o" ]
```



Круглые скобки (..) вокруг `matches = str.match..` обязательны. Это связано с приоритетом операторов, который будет рассматриваться в следующей главе.

Лично я предпочитаю более короткий стиль. Он более четко показывает, что две условные конструкции в действительности связаны, а не существуют по отдельности. Но, как и с большинством стилистических решений в JS, выбор определяется исключительно личными предпочтениями.

Правила контекста

В правилах грамматики JavaScript есть несколько мест, в которых один синтаксис может иметь разный смысл в зависимости от того, где и как используется. Подобные неоднозначности, рассматриваемые вне контекста, могут создать основательную путаницу.

Я не стану приводить здесь полный список, а лишь выделю несколько распространенных случаев.

Фигурные скобки

Есть два основных случая (хотя по мере развития JS их станет больше!), в которых в вашем коде может встретиться пара фигурных скобок. Рассмотрим каждый из них.

Объектные литералы

Прежде всего, это объектный литерал:

```
// предполагается, что функция `bar()` определена
var a = {
  foo: bar()
};
```

Как мы узнаем, что это объектный литерал? Потому что пара фигурных скобок `{ .. }` — это значение, присваиваемое `a`.



Ссылка `a` называется левосторонним значением, потому что она играет роль приемника для присваивания. Пара `{ .. }` называется правосторонним значением, поскольку используется просто как значение (в данном случае как источник присваивания).

Метки

Что произойдет, если убрать из приведенного выше часть `var a = ?`

```
// предполагается, что функция `bar()` определена
{
  foo: bar()
}
```

Многие разработчики считают, что пара `{ .. }` — просто автономный объектный литерал, который ничему не присваивается. Это совершенно не так.

Здесь `{ .. }` — обычный программный блок. В JavaScript подобные автономные блоки `{ .. }` выглядят не особо идиоматично (а уж в других языках и подавно), но это абсолютно законная грамматика JS. Блоки бывают особенно полезны в сочетании с объявлениями с блочной областью видимости.

С функциональной точки зрения программный блок { ... } почти идентичен программному блоку, присоединенному к другой команде: циклу `for/while`, условной команде `if` и т. д.

Но если это обычный блок кода, что это за странный синтаксис `foo: bar()` и как он может быть допустимым?

Все происходит из-за малоизвестной (и честно говоря, не рекомендуемой к использованию) возможности JavaScript, так называемых «команд с метками». `foo` — метка для команды `bar()` (без завершающего символа ; — см. раздел «Автоматические завершители»). Но в чем смысл команды с метками?

Если бы в JavaScript была команда `goto`, теоретически можно было бы использовать команду `goto foo` и передать управление в указанную точку кода. Команды `goto` обычно считаются ужасной идиомой, которая сильно усложняет понимание кода («спагетти-код»), поэтому *очень хорошо*, что в JavaScript нет обобщенной конструкции `goto`.

Однако JS поддерживает ограниченную специальную форму `goto`. При выполнении команд `continue` и `break` может быть указана метка, в этом случае логика программы выполняет «переход» по аналогии с `goto`. Пример:

```
// цикл с меткой `foo`
foo: for (var i=0; i<4; i++) {
    for (var j=0; j<4; j++) {
        // при совпадении переменных циклов продолжить
        // внешний цикл
        if (j == i) {
            // перейти к следующей итерации цикла
            // с меткой `foo`
            continue foo;
        }

        // пропускать нечетные произведения
        if ((j * i) % 2 == 1) {
            // обычное (без метки) продолжение внутреннего цикла
            continue;
        }
    }
}
```

```
        }  
  
        console.log( i, j );  
    }  
}  
// 1 0  
// 2 0  
// 2 1  
// 3 0  
// 3 2
```



`continue foo` не означает «перейти к позиции с меткой `foo`, чтобы продолжить». Смысл другой: «продолжить цикл с меткой `foo` и выполнить его следующую итерацию». Таким образом, это не произвольный переход `goto`.

Как видите, итерация с нечетным произведением $3 \cdot 1$ пропускается, но переход с помеченным циклом также пропустил итерации $1 \cdot 1$ и $2 \cdot 2$.

Возможно, чуть более полезная форма цикла перехода в цикле с меткой — это команда `break __` из внутреннего цикла, когда вы хотите выйти из внешнего цикла. Без `break` с меткой реализация той же логики была бы громоздкой и некрасивой:

```
// цикл с меткой `foo`  
foo: for (var i=0; i<4; i++) {  
    for (var j=0; j<4; j++) {  
        if ((i * j) >= 3) {  
            console.log( "stopping!", i, j );  
            break foo;  
        }  
  
        console.log( i, j );  
    }  
}  
// 0 0  
// 0 1  
// 0 2  
// 0 3
```

```
// 1 0
// 1 1
// 1 2
// stopping! 1 3
```



`break` `foo` не означает «перейти к позиции с меткой `foo`, чтобы продолжить». Смысл другой: «выйти из цикла/блока с меткой `foo` и продолжить выполнение после него». Не похоже на `goto` в традиционном смысле, да?

Вероятно, альтернативная реализация `break` без метки потребует одной или нескольких функций доступа к переменным в общей области видимости и т. д. Скорее всего, она создаст больше путаницы, чем `break` с меткой, поэтому `break` с меткой, пожалуй, будет лучшим решением.

Метка может создаваться в блоке без цикла, но только `break` может ссылаться на такую метку без цикла. Команда `break` — с меткой может осуществлять выход из любого блока с меткой, но выполнить `continue` — с меткой без цикла нельзя, как нельзя и выполнить `break` без метки из блока:

```
// цикл с меткой `bar`
function foo() {
  bar: {
    console.log( "Hello" );
    break bar;
    console.log( "never runs" );
  }
  console.log( "World" );
}

foo();
// Hello
// World
```

Циклы/блоки с метками встречаются крайне редко, и многие разработчики их не одобряют. Лучше избегать их, если это воз-

можно (например, используя вызовы функций вместо переходов из циклов). Наверное, можно придумать несколько особых ситуаций, в которых они могут быть полезными. Если вы собираетесь использовать переход с меткой, обязательно документируйте происходящее в подробных комментариях!

Очень часто встречается мнение, будто JSON является подмножеством JS, так что строка JSON (например, `{"a":42}`), обратите внимание на кавычки вокруг имени свойства, как требует JSON) рассматривается как действительная программа JavaScript. Неправда! Попробуйте ввести `{"a":42}` в консоли JS, и вы получите сообщение об ошибке.

Дело в том, что метки команд не могут заключаться в кавычки, поэтому `"a"` не является действительной меткой, а следовательно, `:` не может следовать после `"a"`. Таким образом, JSON действительно является подмножеством синтаксиса JS, но сам по себе не является действительной грамматикой JS.

Еще одно в высшей степени распространенное заблуждение из той же серии, что если вы загрузите в теге `<script src=..>` файл JS, который содержит только контент JSON (например, полученный при вызове API), данные будут прочитаны как действительный код JavaScript, но будут недоступны программе. Считается, что JSON-P (практика упаковки данных JSON в вызов функции, как в `foo({ "a":42 })`) решает эту проблему недоступности, передавая значение одной из функций вашей программы.

Неправда! Абсолютно законное значение JSON `{"a":42}` само по себе выдаст ошибку JS, потому что будет интерпретировано как блок с недействительной меткой. С другой стороны, `foo({ "a":42 })` является действительным кодом JS, потому что `{"a":42}` интерпретируется как объектный литерал, передаваемый `foo(...)`. Таким образом, можно сказать, что *JSON-P превращает JSON в действительную грамматику JS*.

Блоки

Другая часто упоминаемая проблема JS (связанная с преобразованием типов — см. главу 4):

```
[ ] + { }; // "[object Object]"  
{ } + [ ]; // 0
```

Похоже, отсюда следует, что оператор + дает разные результаты в зависимости от того, какой операнд является первым, [] или { }. Ничего подобного!

В первой строке { } входит в выражение оператора +, а следовательно, интерпретируется как фактическое значение (пустой объект). В главе 4 объясняется, что [] преобразуется в "", а следовательно, { } также преобразуется в строковое значение: "[object Object]".

Но во второй строке { } интерпретируется как автономный пустой блок (который не делает ничего). Блоки не обязаны завершаться символом ;, так что отсутствие завершителя проблем не создает. Наконец, + [] — выражение, которое явно преобразует (см. главу 4) [] в число, отсюда значение 0.

Деструктуризация объектов

Начиная с ES6 пары { .. } также могут встретиться при выполнении «деструктурирующего присваивания», а конкретно при деструктуризации объектов. Пример:

```
function getData() {  
    // ..  
    return {  
        a: 42,  
        b: "foo"  
    };  
}  
  
var { a, b } = getData();  
  
console.log( a, b ); // 42 "foo"
```

Как вы, вероятно, догадались, `var { a, b } = ..` – форма деструктурирующего присваивания ES6, которая приблизительно эквивалентна следующей серии команд:

```
var res = getData();
var a = res.a;
var b = res.b;
```



{ a, b } в действительности является сокращенной деструктурирующей записью ES6 для { a: a, b: b }, так что подойдет любой вариант, но можно ожидать, что более короткая форма { a, b } станет предпочтительной.

Деструктуризация объектов парой { .. } также может использоваться для именованных аргументов функций, удобная запись для аналогичного неявного присваивания значений свойств:

```
function foo({ a, b, c }) {
  // не нужно:
  // var a = obj.a, b = obj.b, c = obj.c
  console.log( a, b, c );
}

foo( {
  c: [1,2,3],
  a: 42,
  b: "foo"
} );    // 42 "foo" [1, 2, 3]
```

Итак, контекст, в котором используются пары { .. }, полностью определяет их смысл, а этот пример демонстрирует различия между синтаксисом и грамматикой. Очень важно понимать эти нюансы, чтобы избежать неожиданных интерпретаций кода движком JS.

Else if и необязательные блоки

Еще одно распространенное заблуждение, что в JavaScript существует конструкция `else if`, так как вы можете сделать следующее:

```
if (a) {  
    // ..  
}  
else if (b) {  
    // ..  
}  
else {  
    // ..  
}
```

Здесь проявляется одна скрытая характеристика грамматики JS: `else if` в языке нет. Но команды `if` и `else` разрешают не заключать присоединенный блок в `{ }` , если они содержат только одну команду. Несомненно, вы уже много раз видели такие команды:

```
if (a) doSomething( a );
```

Многие руководства по стилю JS настаивают, что блоки из одной команды всегда должны заключаться в `{ }` :

```
if (a) { doSomething( a ); }
```

Однако точно такое же правило грамматики применимо к секции `else`, поэтому форма `else if`, которую вы почти наверняка использовали в своих программах, *на самом деле* обрабатывается в следующем виде:

```
if (a) {  
    // ..  
}  
else {  
    if (b) {  
        // ..  
    }  
    else {  
        // ..  
    }  
}
```

`if (b) { .. } else { .. }` — одна команда, следующая за `else`, поэтому вы можете либо заключить ее в `{ }` , либо не заключать.

Иначе говоря, при использовании `else if` вы формально нарушаете это распространенное правило оформления кода и просто определяете собственную секцию `else` из одной команды `if`.

Конечно, идиома `else if` чрезвычайно популярна; вдобавок она сокращает отступы в коде на один уровень. Какой бы способ вы ни выбрали, явно опишите его в своем руководстве по стилю/системе правил и не предполагайте, что такие конструкции, как `else if`, являются прямыми правилами грамматики.

Приоритет операторов

Как упоминалось в главе 4, версии операторов `&&` и `||` в языке JavaScript интересны тем, что они выбирают и возвращают один из операндов (вместо того, чтобы возвращать `true` или `false`). В логике легко разобраться, если команда содержит два операнда и один оператор:

```
var a = 42;
var b = "foo";

a && b; // "foo"
a || b; // 42
```

А если задействованы два оператора и три операнда?

```
var a = 42;
var b = "foo";
var c = [1,2,3];

a && b || c; // ???
a || b && c; // ???
```

Чтобы понять, какой результат дают эти выражения, необходимо понять, по каким правилам должны обрабатываться операторы, входящие в выражение.

Эти правила называются «приоритетом операторов».

Готов поспорить, что многие читатели считают, что они хорошо разбираются в приоритете операторов. Но, как и во всех остальных ситуациях, рассмотренных в этой серии книг, мы проверим это понимание на прочность и выясним, насколько оно прочно в действительности, и возможно, попутно узнаем кое-что новое.

Вспомните приведенный ранее пример:

```
var a = 42, b;  
b = ( a++, a );  
  
a; // 43  
b; // 43
```

Но что произойдет, если удалить ()?

```
var a = 42, b;  
b = a++, a;  
  
a; // 43  
b; // 42
```

Постойте! Почему изменилось значение, присвоенное b?

Потому что оператор , имеет более низкий приоритет, чем оператор =. Таким образом, b = a++, a интерпретируется как (b = a++), a. Потому что (как объяснялось ранее) a++ имеет *побочные эффекты*, значение 42 будет присвоено переменной b до того, как ++ изменит a.

И это очень простой пример, наглядно демонстрирующий важность понимания приоритета операторов. Если вы собираетесь использовать , как оператор серии команд, важно знать, что этот оператор имеет наименьший приоритет. Все остальные операторы будут обрабатываться до , .

А теперь вспомните приведенный выше пример:

```
if (str && (matches = str.match( /[aeiou]/g ))) {  
    // ..  
}
```

Я сказал, что круглые скобки () вокруг присваивания обязательны, но почему? Потому что **&&** обладает более высоким приоритетом, чем **=**, так что без круглых скобок (), определяющих принудительную группировку, выражение будет рассматриваться как **(str && matches) = str.match...** Но это будет ошибкой, потому что результат **(str && matches)** будет не переменной, а значением (в данном случае **undefined**), которое не может находиться в левой части присваивания **=**!

Хорошо. Вероятно, вы думаете, что уж теперь вы точно разбираетесь в приоритетах.

Рассмотрим более сложный пример (к которому мы будем возвращаться в нескольких ближайших разделах), чтобы *понастоящему* проверить ваше понимание:

```
var a = 42;  
var b = "foo";  
var c = false;  
  
var d = a && b || c ? c || b ? a : c && b : a;  
  
d;      // ??
```

Ладно, признаю, никто не будет включать в свои программы такие цепочки выражений. *Скорее всего* так, но мы используем этот пример для изучения различных аспектов сцепления операторов — очень распространенной задачи.

Результат равен 42. Но здесь интересно вовсе не это, а то, как вычислить этот ответ без того, чтобы запустить JS-программу и поручить JavaScript разобраться во всем.

Первый вопрос, который, возможно, даже не пришел вам в голову: как себя ведет первая часть (`a && b || c`), как (`a && b`) `||` `c` или как `&& (b || c)`? Вы в этом уверены? А вы вообще уверены в том, что они действительно различны?

```
(false && true) || true;      // true
false && (true || true);    // false
```

Да, это доказывает, что они различны. Но как себя ведет конструкция `false && true || true`? Ответ:

```
false && true || true;      // true
(false && true) || true;    // true
```

Итак, оператор `&&` вычисляется первым, а оператор `||` вычисляется вторым.

Но может, это объясняется обработкой слева направо? Давайте изменим порядок операторов:

```
true || false && false;    // true
(true || false) && false;    // false - нет
true || (false && false);  // true - победа!
```

Теперь мы убедились в том, что `&&` вычисляется первым, а после него вычисляется `||`; в данном случае это противоречит обычно ожидаемой обработке слева направо.

Что же вызвало это поведение? *Приоритет операторов*. Но меня сильно удручает, как мало разработчиков JS действительно прочитали этот список.

Если бы вы знали его действительно хорошо, эти примеры не вызвали бы у вас ни малейших затруднений, потому что вы бы уже знали, что `&&` обладает более высоким приоритетом, чем `||`. Но я уверен, что многим читателям пришлось хотя бы ненадолго задуматься.



К сожалению, в спецификации JS список приоритета операторов не собран в одном удобном месте. Вам придется разбираться в описаниях и понять все правила грамматики. По этой причине мы попробуем представить самые распространенные и полезные случаи в более удобном формате. За полным списком приоритета операторов обращайтесь к разделу «Приоритет операторов» на сайте MDN¹.

Ускоренная обработка

В главе 4 упоминалось о механизме ускоренной обработки, используемом для таких операторов, как `&&` и `||`. Вернемся к этой теме и рассмотрим ее более подробно.

Для операторов `&&` и `||` правый operand не вычисляется, если operand в левой части достаточно для определения результата операции. Отсюда и название «ускоренная обработка» (то есть обработка, которая может завершиться преждевременно).

Например, с `a && b` переменная `b` не вычисляется, если значение `a` ложно — ведь результат operand `&&` уже определен, поэтому возиться с проверкой `b` не обязательно. Аналогичным образом, если в выражении `a || b` переменная `a` истинна, результат оператора уже определен, поэтому проверять `b` уже не нужно. Ускоренная обработка бывает очень полезной и часто применяется на практике:

```
function doSomething(opts) {  
  if (opts && opts.cool) {  
    // ..  
  }  
}
```

Часть `opts` проверки `opts && opts.cool` действует как своего рода защита, потому что если значение `opts` не задано (или не явля-

¹ https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Operator_Precidence.

ется объектом), выражение `opts.cool` выдаст ошибку. Не прошедшая проверка `opts` в сочетании с ускоренной обработкой означает, что значение `opts.cool` вычисляться не будет, а следовательно, не будет и ошибки!

Аналогичным образом можно использовать ускоренную обработку `||`:

```
function doSomething(opts) {  
  if (opts.cache || primeCache()) {  
    // ..  
  }  
}
```

Здесь мы сначала проверяем `opts.cache`, и если значение присутствует, то функция `primeCache()` не вызывается, чтобы избежать лишней работы.

Плотное связывание

Но вернемся к предыдущему примеру сложной команды со сцепленными операторами, а конкретно к тернарным операторам `? :`. Обладает ли оператор `? :` большим или меньшим приоритетом, чем операторы `&&` и `||`?

```
a && b || c ? c || b ? a : c && b : a
```

На что это больше похоже — на это:

```
a && b || (c ? c || (b ? a : c) && b : a)
```

Или на это:

```
(a && b || c) ? (c || b) ? a : (c && b) : a
```

Правильный ответ второй. Но почему?

Потому что `&&` обладает более высоким приоритетом, чем `||`, а `||` — более высоким приоритетом, чем `? :`.

Таким образом, выражение `(a && b || c)` будет вычислено до вычисления оператора `? :`, в котором оно участвует. Также часто говорят, что `&&` и `||` «связываются плотнее», чем `? :`. Если бы было справедливо обратное, то `c ? c..` были бы связаны плотнее, и вся конструкция обладала бы поведением первого варианта — `a && b || (c ? c..)`.

Ассоциативность

Итак, сначала связываются операторы `&&` и `||`, а потом оператор `? :`. Но как насчет нескольких операторов с одинаковым приоритетом? Всегда ли они обрабатываются слева направо или справа налево?

В общем случае операторы обладают левосторонней или правосторонней ассоциативностью в зависимости от того, происходит ли группировка слева направо или справа налево.

Важно заметить, что ассоциативность — не то же самое, что обработка слева направо или справа налево.

Но почему важно, выполняется ли обработка слева направо или справа налево? Потому что выражения могут иметь побочные эффекты, например вызовы функций:

```
var a = foo() && bar();
```

Здесь сначала вычисляется `foo()`, а затем может вычисляться часть `bar()`, которая может зависеть от результата выражения `foo()`. Это определенно изменит поведение программы, в отличие от того, как если бы функция `bar()` была вызвана перед `foo()`.

Но это поведение — обычная обработка слева направо (поведение по умолчанию в JavaScript), и оно не имеет никакого отношения к ассоциативности `&&`. Поскольку в данном примере используется только один оператор `&&` (а следовательно, группировка отсутствует), ассоциативность вообще не учитывается.

Но в выражениях вида `a && b && c` будет выполнена неявная группировка; это означает, что либо выражение `a && b`, либо `b && c` будет выполнено первым.

С технической точки зрения `a && b && c` будет обрабатываться как `(a && b) && c`, потому что оператор `&&` обладает левосторонней ассоциативностью (как и `||`, кстати говоря). Однако альтернатива с правосторонней ассоциативностью `a && (b && c)` ведет себя во всех отношениях так же. Для тех же значений те же выражения будут вычисляться в том же порядке.



Гипотетически если бы оператор `&&` обладал правосторонней ассоциативностью, он обрабатывался бы так же, как если бы вы вручную использовали `()` для группировки вида `&& (b && c)`. Но и это не означает, что `c` будет обрабатываться раньше `b`. Правосторонняя ассоциативность не означает, что вычисление будет происходить справа налево. Она означает, что группировка будет происходить справа налево. В любом случае независимо от группировки/ассоциативности вычисления будут происходить строго в порядке: `a`, затем `b`, затем `c` (слева направо).

Итак, на самом деле не так уж важно, что операторы `&&` и `||` обладают левосторонней ассоциативностью, разве что для точности обсуждения их определений.

Но это не всегда так. Некоторые операторы обладают очень разным поведением в зависимости от левосторонней или правосторонней ассоциативности. Возьмем оператор `? :` (тернарный) оператор:

```
a ? b : c ? d : e;
```

Оператор `? :` обладает правосторонней ассоциативностью; какая группировка правильно описывает последовательность его обработки?

- `a ? b : (c ? d : e)`
- `(a ? b : c) ? d : e`

Правильный ответ — `a ? b : (c ? d : e)`. В отличие от приведенных выше операторов `&&` и `||`, правосторонняя ассоциативность здесь действительно важна, поскольку `(a ? b : c) ? d : e` *будет* вести себя по-разному для некоторых (но не для всех!) комбинаций значений.

Один из примеров такого рода:

```
true ? false : true ? true : true;      // false
true ? false : (true ? true : true);    // false
(true ? false : true) ? true : true;    // true
```

Еще более тонкие различия кроются в других комбинациях значений, хотя конечный результат остается неизменным. Пример:

```
true ? false : true ? true : false;     // false
true ? false : (true ? true : false);  // false
(true ? false : true) ? true : false; // false
```

В этом сценарии тот же конечный результат подразумевает, что группировка неважна. Но при этом:

```
var a = true, b = false, c = true, d = true, e = false;
a ? b : (c ? d : e); // false, вычисляет только `a` и `b`
(a ? b : c) ? d : e; // false, вычисляет только `a`, `b` и `e`
```

Итак, мы очевидно показали, что оператор `? :` обладает правосторонней ассоциативностью и что на самом деле это важно в отношении того, как оператор ведет себя при сцеплении с самим собой.

Другой пример правосторонней ассоциативности (группировки) — оператор `=`. Вспомните пример сцепленного присваивания, приведенный ранее в этой главе:

```
var a, b, c;  
a = b = c = 42;
```

Ранее мы предположили, что для обработки `a = b = c = 42` сначала вычисляется присваивание `c = 42`, затем `b = ..`, и наконец `a = ...`. Почему?

Из-за правосторонней ассоциативности, которая фактически интерпретирует команду следующим образом: `a = (b = (c = 42))`.

Помните сложный пример с присваиванием, приведенный ранее в этой главе?

```
var a = 42;  
var b = "foo";  
var c = false;  
  
var d = a && b || c ? c || b ? a : c && b : a;  
  
d; // 42
```

Вооружившись новыми знаниями о приоритете и ассоциативности, мы теперь сможем разбить код в соответствии с группировкой поведения:

```
((a && b) || c) ? ((c || b) ? a : (c && b)) : a
```

То же с отступами, чтобы было проще понять:

```
(  
  (a && b)  
  ||  
  c  
)  
 ?  
(  
  (c || b)  
  ?  
  a  
  :  
  )
```

```
(c && b)  
)  
:  
a
```

А теперь проведем вычисления:

1. (a && b) дает "foo".
2. "foo" || c дает "foo".
3. Для первой проверки ? "foo" является истинным.
4. (c || b) дает "foo".
5. Для второй проверки ? "foo" является истинным.
6. a равно 42.

Вот и все! Ответ 42, как и было показано ранее. Не так уж сложно, верно?

Неоднозначности

Вероятно, к этому моменту вы уже намного лучше понимаете приоритет (и ассоциативность) операторов и понимаете, как будет вести себя код с несколькими сцепленными операторами.

Но остается один важный вопрос: должны ли все мы писать код с полным пониманием правил приоритета/ассоциативности операторов? Должны ли мы использовать ручную группировку () только тогда, когда необходимо определить другой порядок обработки (связывание)?

Или же следует признать, что хотя эти правила *можно* освоить, в них кроется достаточно ловушек, чтобы от автоматических приоритетов/ассоциативности держаться подальше? А если так, не следует ли всегда использовать ручную группировку ()

и полностью отказаться от использования этого автоматического поведения в программе?

Эта тема в высшей степени субъективна и сильно напоминает обсуждение неявного преобразования из главы 4. Многие разработчики занимают похожую позицию по обоим вопросам: либо они принимают оба аспекта поведения и программируют, рассчитывая на них, либо отказываются от обоих и придерживаются явных/ручных идиом.

Конечно, я не могу дать читателю однозначный ответ на этот вопрос (как, впрочем, и было в главе 4). Но я представил все плюсы и минусы и, надеюсь, помог достаточно глубоко разобраться в теме, чтобы вы могли принимать решения обоснованно, а не под влиянием эмоций.

На мой взгляд, существует важная промежуточная позиция. В программах стоит использовать сочетание приоритета/ассоциативности *и* ручной группировки () — точно так же в главе 4 я выступаю за здоровое/безопасное применение неявного преобразования, но, безусловно, не выступаю за то, чтобы применять его повсеместно и неограниченно.

Например, конструкция `if (a && b && c) ..` кажется мне абсолютно нормальной, и я не стану переписывать ее в виде `if ((a && b) && c) ..` просто для того, чтобы явно выделить ассоциативность, потому что, на мой взгляд, это излишне.

С другой стороны, если мне потребуется сцепить два тернарных оператора `? :`, я, безусловно, использую ручную группировку (), чтобы абсолютно четко обозначить свою предполагаемую логику.

Таким образом, мой совет сведен с советом из главы 4: используйте приоритет/ассоциативность операторов там, где это приводит к более компактному и ясному коду, но используйте явную группировку () в тех местах, где это поможет создать более ясный код и устранит любую путаницу.

Автоматические точки с запятой

Термин ASI (Automatic Semicolon Insertion) означает, что в некоторых местах вашей JS-программы JavaScript предполагает наличие символа ;, даже если вы этот символ туда не поставили.

Зачем это нужно? Потому что если вы опустите хотя бы один обязательный символ ;, ваша программа работать не будет. Сурово, да? ASI позволяет JS нормально переносить некоторые ситуации, в которых многие разработчики считают, что присутствие символа ; не обязательно.

Важно заметить, что ASI работает только в позиции новой строки (разрыва строки). Символ ; не вставляется в середине строки.

По сути, если парсер JS разбирает строку, в которой должна произойти ошибка разбора (отсутствие ожидаемого символа ;), и в этой строке можно разумно вставить такой символ, он так и делает. Какую позицию считать «разумной» для вставки? Только если между концом некоторой команды и символом новой строки в этой строке кода нет ничего, кроме пробелов и/или комментариев.

Пример:

```
var a = 42, b
c;
```

Следует ли JS рассматривать с в следующей строке как часть команды var? Несомненно, парсер так бы и поступил, если бы между b и с где-то (даже в другой строке) стояла запятая (,). Но так как запятой нет, JS считает, что после b (перед новой строкой) подразумевается символ ;. Таким образом, c; остается как автономная команда-выражение.

Аналогичный пример:

```
var a = 42, b = "foo";
```

```
a
b // "foo"
```

Этот фрагмент остается допустимой программой без ошибок, потому что команды-выражения также поддерживаютASI.

В некоторых местахASI приносит пользу:

```
var a = 42;

do {
    // ..
} while (a) // <-- здесь ожидается символ ;
a;
```

Грамматика требует наличия ; после цикла `do..while`, но не после циклов `while` или `for`. Но многие разработчики этого не помнят!.ASI приходит на помощь и вставляет этот символ.

Как говорилось ранее в этой главе, блоки команд не требуют завершителя ;, так что вмешательствоASI не обязательно:

```
var a = 42;

while (a) {
    // ..
} // <-- здесь не ожидается символ ;
a;
```

Еще одна ситуация, в которой требуется вмешательствоASI, – ключевые слова `break`, `continue`, `return` и `yield` (ES6):

```
function foo(a) {
    if (!a) return
    a *= 2;
    // ..
}
```

Команда `return` не переходит на новую строку к выражению `a *= 2`, так как ASI учитывает символ `;`, завершающий команду `return`. Конечно, команды `return` легко *могут* распространяться на несколько строк, но только не в тех случаях, когда после `return` нет ничего, кроме новой строки:

```
function foo(a) {  
    return (  
        a * 2 + 3 / 12  
    );  
}
```

Идентичные рассуждения относятся к `break`, `continue` и `yield`.

Исправление ошибок

Одна из самых яростных идеиных войн в сообществе JS (если не считать войны табуляций против пробелов) идет по поводу того, стоит ли в значительной мере/исключительно полагаться на ASI.

Чаще всего, хотя и не всегда, символы `;` не являются обязательными, но два символа `;` в заголовке цикла `for (..) ..` обязательны.

Аргументы «за»: многие разработчики считают, что ASI — положительный механизм, который позволяет писать более компактный (а следовательно, более «красивый») код, разработчик может опустить все символы `;`, кроме абсолютно необходимых (которых очень мало). Часто считается, что с ASI многие `;` становятся необязательными, поэтому правильно написанная программа *без них* не отличается от правильно написанной программы *с ними*.

Аргументы «против»: многие другие разработчики считают, что в программе *слишком много* мест, в которых ASI может создать случайные ловушки, где волшебным образом вставленные `;` случайно изменяют смысл, особенно для новых, менее опытных

разработчиков. Также некоторые разработчики считают, что если они пропустили точку с запятой — это несомненная ошибка, и их инструментарии (статические анализаторы и т. д.) должны обнаружить эту ошибку до того, как движок JS незаметно ее исправит.

Позвольте поделиться моей точкой зрения. В спецификации буквально сказано, что ASI является инструментом «исправления ошибок». Каких ошибок, спросите вы? *Ошибка разбора*. Другими словами, стараясь уменьшить количество ошибок парсера, ASI позволяет ему ослабить бдительность.

Но бдительность к чему? В моем представлении ошибка разбора происходит только в том случае, если парсер получает для разбора некорректную/ошибочную программу. Итак, хотя механизм ASI строго исправляет ошибки разбора, он может получить такие ошибки только в том случае, если программа изначально содержит ошибки разработки — пропущенные символы ; там, где они необходимы по правилам грамматики.

Проще говоря, когда я слышу, что кто-то предпочитает «опускать необязательные точки с запятой», я воспринимаю это как «хочу написать самую некорректную с точки зрения парсера программу, которая все еще будет работать».

Это совершенно смехотворная позиция, а аргументы о сэкономленных нажатиях клавиш и более «красивом коде» выглядят в лучшем случае слабо.

Более того, я не могу согласиться с тем, что этот спор равнозначен спору о пробелах и табуляциях (в этом случае различия часто косметические). Я считаю, что это фундаментальный вопрос о написании кода, соответствующего требованиям грамматики, против кода, использующего грамматические исключения, чтобы быть хоть сколько-нибудь работоспособным.

Также можно взглянуть на проблему под другим углом: полагаться на ASI фактически означает считать новые строки значи-

мыми «пропусками». В других языках (например, в Python) действительно существуют значимые пропуски. Но насколько уместно рассматривать JavaScript как язык со значимыми пропусками в его нынешнем состоянии?

Моя точка зрения: используйте символы ; везде, они заведомо «обязательны», и сведите количество предположений обASI к минимуму.

Не предлагаю верить мне на слово. В 2012 году Брэндан Эйх (Brendan Eich), создатель JavaScript, сказал следующее¹:

«Мораль:ASI (формально говоря) является процедурой исправления синтаксических ошибок. Если вы начнете программировать так, словно существует универсальное правило о значимых новых строках, то сами напрашиваетесь на проблемы... Мне бы хотелось сделать новые строки более значимыми в JS в те десять дней в мае 1995 года... Страйтесь не использоватьASI, потому что он фактически вводит в JS значимые новые строки».

Ошибки

В JavaScript существуют разные *подтипы* ошибок (`TypeError`, `ReferenceError`, `SyntaxError` и т. д.), и грамматика определяет некоторые ошибки, которые должны выявляться на стадии компиляции — в отличие от ошибок, возникающих на стадии выполнения.

В частности, давно существовали различные конкретные условия, которые должны обнаруживаться как «ранние ошибки» (на стадии компиляции). Любые очевидные ошибки синтаксиса (например, `a =`) относятся к категории ранних, но грамматика

¹ <https://brendaneich.com/2012/04/the-infernal-semicolon/>.

также определяет некоторые конструкции, которые допустимы с позиций синтаксиса, но все равно запрещены.

Так как выполнение кода еще не началось, эти ошибки не могут быть перехвачены конструкцией `try..catch`; вместо этого они просто нарушают разбор кода/компиляцию программы.



В спецификации нет требований относительно того, как именно браузеры (и средства разработчика) должны сообщать об ошибках. Возможно, в следующих примерах вам встретятся некоторые отличия в зависимости от браузеров: в конкретных подтипах ошибок или в тексте сообщений об ошибках.

Простой пример — синтаксис литералов регулярных выражений. В следующей команде с синтаксисом JS все нормально, но недействительное регулярное выражение приводит к выдаче ранней ошибки:

```
var a = /+foo/;      // Ошибка!
```

Приемником присваивания должен быть идентификатор (или деструктурирующее выражение ES6, производящее один или несколько идентификаторов), так что такое значение, как `42`, в этой позиции недопустимо, и об ошибке можно сообщить немедленно:

```
var a;  
42 = a;      // Ошибка!
```

Строгий режим ES5 определяет еще больше ранних ошибок. Например, в строгом режиме имена параметров функций не могут дублироваться:

```
function foo(a,b,a) { }          // нормально  
function bar(a,b,a) { "use strict"; } // Ошибка!
```

Другая ранняя ошибка строкового режима — объектный литерал с несколькими одноименными свойствами:

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

  var a = {
    b: 42,
    b: 43
  };           // Ошибка!
})();
```



С точки зрения синтаксиса такие ошибки не являются *синтаксическими*. Они ближе к грамматическим ошибкам — все приведенные выше фрагменты являются синтаксически действительными. Но поскольку отдельного типа `GrammarError` не существует, некоторые браузеры используют вместо него `SyntaxError`.

Преждевременное использование переменных

В ES6 определяется новая концепция «временной мертвой зоны» TDZ (Temporal Dead Zone) — честно говоря, название только запутывает.

Термином «TDZ» обозначаются те места в вашем коде, в которых к переменной еще нельзя обратиться, потому что она еще не достигла точки необходимой инициализации. Самый очевидный пример — блочная область видимости ES6 `let`:

```
{
  a = 2;      // ReferenceError!
  let a;
}
```

Присваивание `a = 2` обращается к переменной (с блочной областью видимости для блока `{ ... }`) до того, как она будет ини-

циализирована объявлением `let a`. Присваивание находится в TDZ для `a` и инициирует ошибку.

Интересно, что хотя у `typeof` имеется исключение безопасности для необъявленных переменных (см. главу 1), для ссылок TDZ такого исключения не существует:

```
{  
  typeof a;    // undefined  
  typeof b;    // ReferenceError! (TDZ)  
  let b;  
}
```

Аргументы функций

Другой пример нарушения TDZ встречается в значениях параметров по умолчанию ES6:

```
var b = 3;  
  
function foo( a = 42, b = a + b + 5 ) {  
  // ..  
}
```

Обращение к `b` в присваивании произойдет в TDZ параметра `b` (не для внешней ссылки `b`), поэтому произойдет ошибка. С другой стороны, с переменной `a` в этом случае все будет нормально, поскольку обращение происходит за пределами TDZ параметра `a`.

При использовании значений параметров по умолчанию ES6 значение по умолчанию применяется к параметру, если аргумент пропущен или же вместо него передается значение `undefined`:

```
function foo( a = 42, b = a + 1 ) {  
  console.log( a, b );  
}  
  
foo();           // 42 43
```

```
foo( undefined );           // 42 43
foo( 5 );                  // 5 6
foo( void 0, 7 );          // 42 7
foo( null );               // null 1
```



null преобразуется в значение 0 в выражении a + 1. За дополнительной информацией обращайтесь к главе 4.

С точки зрения значений параметров по умолчанию ES6 между пропущенным аргументом и передачей значения undefined нет никаких различий. Однако в некоторых случаях различия все же удается выявить:

```
function foo( a = 42, b = a + 1 ) {
  console.log(
    arguments.length, a, b,
    arguments[0], arguments[1]
  );
}

foo();                      // 0 42 43 undefined undefined
foo( 10 );                  // 1 10 11 10 undefined
foo( 10, undefined );      // 2 10 11 10 undefined
foo( 10, null );           // 2 10 null 10 null
```

Хотя к параметрам a и b применяются значения параметров по умолчанию, если в этих позициях не передаются аргументы, массив arguments не будет содержать элементов.

И наоборот, если явно передать аргумент undefined, в массиве arguments будет присутствовать элемент для этого аргумента. При этом он будет равен undefined, а не значению по умолчанию, которое было применено к именованному параметру в этой позиции (хотя и не обязательно).

Хотя значения параметров по умолчанию ES6 могут создать несоответствие между элементом массива arguments и соответствую-

ющей переменной именованного параметра, такие же несоответствия могут нетривиально проявиться и в ES5:

```
function foo(a) {  
  a = 42;  
  console.log( arguments[0] );  
}  
  
foo( 2 );    // 42 (связь есть)  
foo();       // undefined (связи нет)
```

Если аргумент передается, то элемент `arguments` и именованный параметр, с которым он связан, всегда содержат одинаковые значения. Если аргумент пропущен, такая связь отсутствует.

Но в строгом режиме связь не существует в любом случае:

```
function foo(a) {  
  "use strict";  
  a = 42;  
  console.log( arguments[0] );  
}  
  
foo( 2 );    // 2 (связь есть)  
foo();       // undefined (связи нет)
```

Любая зависимость от таких связей почти наверняка нежелательна. Собственно, сама связь скорее выглядит как «дырявая абстракция», которая открывает доступ к подробностям реализации движка, а не как нормально спроектированная возможность.

Массив `arguments` считается устаревшим, и использовать его не рекомендуется (особенно с параметрами ... из ES6 — см. книгу «ES6 и далее» этой серии). Тем не менее это не означает, что массив совершенно плох.

До выхода ES6 массив `arguments` был единственным способом получения массива всех переданных аргументов для передачи другим функциям, что, как оказалось, было довольно полезно. Вы также можете объединить именованные параметры с массиви-

вом `arguments` и обеспечить безопасность при условии соблюдения одного простого правила: *никогда не обращаться к именованному параметру и соответствующему элементу arguments одновременно*. При выполнении этого условия ненадежное поведение связей никогда не проявится в программе:

```
function foo(a) {  
    console.log( a + arguments[1] ); // безопасно!  
}  
  
foo( 10, 32 ); // 42
```

try..finally

Вероятно, вы знаете, как работает блок `try..catch`. Но вы когда-нибудь задумывались над секцией `finally`, которая может связываться с этим блоком? А вы вообще знали, что `try` требует только одной из секций — `catch` или `finally`, хотя при необходимости могут присутствовать обе секции?

Код в секции `finally` выполняется *всегда* (независимо от любых обстоятельств), и он всегда выполняется сразу же после завершения `try` (и `catch`, если она присутствует), перед выполнением любого другого кода. В каком-то смысле можно рассматривать код `finally` как функцию обратного вызова, которая всегда выполняется независимо от поведения остальной части блока.

Что же произойдет, если в секции `try` присутствует ключевое слово `return`? Очевидно, она вернет значение, правильно? Но будет ли вызывающий код, получивший значение, выполнен до или после `finally`?

```
function foo() {  
    try {  
        return 42;  
    }  
}
```

```
finally {
  console.log( "Hello" );
}

console.log( "never runs" );
}

console.log( foo() );
// Hello
// 42
```

Команда `return 42` выполняется немедленно, и в ней определяется возвращаемое значение вызова `foo()`. Это действие завершает секцию `try`, после чего немедленно выполняется функция `finally`. Только после этого функция `foo()` по-настоящему завершится, а ее возвращаемое значение будет передано команде `console.log(..)` для использования.

Точно так же ведет себя `throw` внутри `try`:

```
function foo() {
  try {
    throw 42;
  }
  finally {
    console.log( "Hello" );
  }

  console.log( "never runs" );
}

console.log( foo() );
// Hello
// Uncaught Exception: 42
```

Если в секции `finally` будет выдано исключение (случайно или намеренно), оно переопределит первичную зависимость этой функции. Если предшествующая команда `return` в блоке `try` задала возвращаемое значение для функции, это значение будет потеряно:

```
function foo() {
    try {
        return 42;
    }
    finally {
        throw "Oops!";
}
console.log( "never runs" );
}

console.log( foo() );
// Uncaught Exception: Упс!
```

Не стоит удивляться тому, что другие нелинейные управляющие команды (такие, как `continue` и `break`) демонстрируют поведение, аналогичное `return` и `throw`:

```
for (var i=0; i<10; i++) {
    try {
        continue;
    }
    finally {
        console.log( i );
    }
}
// 0 1 2 3 4 5 6 7 8 9
```

Команда `console.log(i)` выполняется в конце цикла, что обусловлено присутствием команды `continue`. Тем не менее она все равно выполняется ранее команды обновления переменной цикла `i++`; это объясняет, почему выводятся значения `0..9` вместо `1..10`.



В ES6 в генераторы была добавлена команда `yield`, которая в некоторых отношениях может рассматриваться как промежуточная команда `return`. Однако в отличие от `return`, `yield` не завершается до возобновления работы генератора; это означает, что конструкция `try { .. yield .. }` не завершилась. Таким образом, присоединенная секция `finally` не будет выполняться сразу же после `yield`, как это происходит с `return`.

Команда `return` внутри `finally` наделена способностью переопределения предыдущей команды `return` из секций `try` или `catch`, но только если `return` вызывается явно:

```
function foo() {  
    try {  
        return 42;  
    }  
    finally {  
        // здесь нет `return ..` , поэтому нет  
        // переопределения  
    }  
}  
  
function bar() {  
    try {  
        return 42;  
    }  
    finally {  
        // переопределяет предыдущую команду  
        // `return 42`  
        return;  
    }  
}  
  
function baz() {  
    try {  
        return 42;  
    }  
    finally {  
        // переопределяет предыдущую команду  
        // `return 42`  
        return "Hello";  
    }  
}  
  
foo(); // 42  
bar(); // undefined  
baz(); // Hello
```

Обычно отсутствующая команда `return` в функции эквивалентна `return;` или даже `return undefined;`, но внутри блока `finally` от-

существие `return` не работает как переопределяющая команда `return undefined`; предыдущая команда просто продолжает действовать.

Более того, градус безумия можно повысить, объединив `finally` с командой `break` с меткой (см. «Метки»):

```
function foo() {
  bar: {
    try {
      return 42;
    }
    finally {
      // выход из блока с меткой `bar`
      break bar;
    }
  }
  console.log( "Crazy" );

  return "Hello";
}

console.log( foo() );
// Crazy
// Hello
```

Но... не надо так делать. Серьезно. Комбинация `finally + break` с меткой для фактической отмены `return` — серьезная заявка на написание самого невразумительного кода из всех возможных. Сколько бы комментариев вы ни написали, это все равно не поможет.

switch

А теперь кратко рассмотрим команду `switch`, своего рода синтаксическое сокращение для цепочки команд `if..else if..else..:`

```
switch (a) {
  case 2:
```

```
// что-то делаем
break;
case 42:
    // делаем что-то другое
    break;
default:
    // для остальных случаев
}
```

Как видите, условие проверяется один раз, а затем полученный результат сравнивается с выражением в каждой секции `case` (здесь размещаются простые выражения). Если совпадение будет найдено, выполнение продолжается с этой секции `case` и продолжается либо до команды `break`, либо до конца блока `switch`.

У команды `switch` есть несколько странностей, которые вы, возможно, не замечали ранее.

Прежде всего, проверка совпадения выражения `a` с каждым выражением в `case` выполняется по правилам алгоритма `==` (см. главу 4). Часто в секциях `case` указываются абсолютные значения, как в приведенном примере, так что строгая проверка равенства уместна.

Однако может оказаться, что для ваших целей больше подходит равенство с преобразованием типов (или `==`, см. главу 4). Чтобы включить его, необходимо слегка «подправить» команду `switch`:

```
var a = "42";

switch (true) {
    case a == 10:
        console.log( "10 or '10'" );
        break;
    case a == 42:
        console.log( "42 or '42'" );
        break;
    default:
        // сюда управление никогда не передается
}
// 42 или '42'
```

Такое решение работает, потому что секция `case` может содержать любое выражение (а не только простые значения); результат этого выражения будет строго сравниваться с проверяемым условием (`true`). Так как `a == 42` дает `true`, совпадение будет успешно обнаружено.

Несмотря на `==`, сама проверка совпадения выполняется строго между `true` и `true` в данном случае. Если выражение `case` дает другое значение — истинное, но отличное от `true` (см. главу 4), совпадения не будет. Этот факт может преподнести неприятный сюрприз, если в выражении, например, используется «логический оператор» `||` или `&&`:

```
var a = "hello world";
var b = 10;

switch (true) {
    case (a || b == 10):
        // сюда управление никогда не передается
        break;
    default:
        console.log( "Oops" );
}
// Не работает
```

Так как выражение `(a || b == 10)` дает результат `"hello world"`, а не `true`, строгая проверка завершается неудачей. В данном случае проблема решается явным приведением выражения к `true` или `false`, например `case !(a || b == 10):` (см. главу 4).

Наконец, присутствие секции `default` не обязательно, и она не обязана располагаться в конце (хотя это очень распространенная схема). Даже в секции `default` действуют те же правила по поводу наличия или отсутствия `break`:

```
var a = 10;

switch (a) {
    case 1:
    case 2:
```

```
// сюда управление никогда не передается
default:
    console.log( "default" );
case 3:
    console.log( "3" );
    break;
case 4:
    console.log( "4" );
}
// default
// 3
```



Как упоминалось ранее относительно `break` с метками, команда `break` в условии `case` тоже может быть снабжена меткой.

В этом фрагменте сначала проверяются условия всех секций `case`, совпадение не обнаруживается. Тогда команда возвращается к секции `default` и начинает выполнять ее. Так как команда `break` здесь отсутствует, выполнение продолжается в блоке уже пропущенной ранее секции 3, и только после достижения команды `break` в этой секции оно прерывается.

Хотя такая «карусель» очевидно возможна в JavaScript, она почти ни при каких условиях не приведет к созданию логичного или понятного кода. Если вам вдруг понадобится реализовать такую круговую логику, вам стоит очень скептично отнестись к такой идее, а если это все же неизбежно, то обязательно включите больше комментариев, поясняющих ваши намерения!

Итоги

Грамматика JavaScript имеет множество нюансов, на которые нам как разработчикам стоило бы обращать чуть больше внимания, чем это обычно бывает. Ценой небольших дополнительных усилий вы сможете сильно укрепить свое знание языка.

У команд и выражений есть аналоги в естественном языке — команды напоминают предложения, а выражения — отдельные фразы. Выражения могут быть «чистыми»/автономными, а могут иметь побочные эффекты.

Грамматика JavaScript накладывает семантические правила использования (то есть контекст) на чистый синтаксис. Например, фигурные скобки `{ }` в разных частях программы могут обозначать блоки команд, объектные литералы, деструктурирующее присваивание (ES6) или именованные аргументы функций (ES6).

Для всех операторов JavaScript определены четкие правила приоритета (для каких операторов связывание выполняется раньше других) и ассоциативности (как выполняется неявная группировка в выражениях с несколькими операторами). Когда вы изучите эти правила, решите сами: то ли приоритет/ассоциативность настолько неявны, что это идет им во вред, то ли они помогают писать более короткий и ясный код.

ASI (Automatic Semicolon Insertion) — механизм исправления ошибок парсера, встроенный в движок JS и позволяющий в некоторых ситуациях вставлять предполагаемые символы `;` в тех местах, где эти символы необходимы, были пропущены *и* где вставка исправит ошибку разбора. По поводу этого поведения идут жаркие споры: по мнению одних, оно подразумевает, что многие символы `;` не обязательны (и их можно/нужно опустить для ясности кода). По мнению других — пропущенные символы `;` суть ошибки, которые движок JS просто исправляет за вас.

JavaScript поддерживает несколько типов ошибок, но существует другая, менее известная классификация ошибок на две категории: «ранние ошибки» (выдаются компилятором и не перехватываются программой) и «ошибок времени выполнения» (для `try..catch`). Разумеется, все синтаксические ошибки относятся к категории ранних, прерывающих работу программы перед ее выполнением, но существуют и другие виды ошибок.

Существует интересная связь аргументов функций с их формально объявленными именованными параметрами. А именно массив `arguments` создает некоторые ловушки из разряда «дырявой абстракции», если вы не будете осторожны. Страйтесь обходиться без массива `arguments`, если можете, но, если вынуждены использовать его — любыми способами избегайте использования позиционных обращений к элементам `arguments` одновременно с именованным параметром того же аргумента.

Секция `finally`, присоединенная к `try` (или `try..catch`), создает несколько очень интересных странностей в отношении порядка выполнения. Одни из этих странностей могут быть полезными, но они также могут создать изрядную путаницу — особенно в сочетании с блоками с метками. Как обычно, `finally` следует использовать для создания кода более качественного и понятного, а не более хитроумного или запутанного.

Команда `switch` предоставляет удобную сокращенную запись для цепочек `if..else if..`, но следует остерегаться многих распространенных упрощающих предположений относительно ее поведения. Да, у нее есть свои проблемы, о которые можно споткнуться при невнимательности, однако у `switch` в запасе также найдется и кое-что полезное!

Приложение А. JavaScript в разных средах

В этой книге были полностью рассмотрены многие базовые механизмы языка. Но при выходе вашего кода JS в реальный мир он может вести себя по-разному в разных условиях. Если JS выполняется исключительно внутри движка, его поведение полностью предсказуемо и определяется исключительно спецификацией и ничем иначе. Однако JS почти всегда работает в контексте среды внешнего размещения, которая создает некоторую степень неопределенности.

Например, когда ваш код выполняется параллельно с кодом из других источников или когда ваш код выполняется в разных типах ядер JS (не только в браузерах), некоторые аспекты поведения могут измениться.

В этом приложении будут кратко рассмотрены некоторые потенциальные изменения.

Дополнение В (ECMAScript)

Малоизвестный факт: язык официально называется ECMAScript (ссылка на комитет по стандартизации ЕССА, управляющего им). Тогда что же такое «JavaScript»? Конечно, JavaScript – рас-

пространенное наименование языка, но правильнее сказать, что JavaScript, по сути, представляет собой браузерную реализацию спецификации.

Официальная спецификация ECMAScript включает «дополнение B», в котором обсуждаются конкретные отклонения от официальной спецификации для обеспечения совместимости JS в браузерах.

Как следует относиться к этим отклонениям? Они актуальны только в том случае, если код выполняется в браузере. Если ваш код всегда работает в браузерах, вы не заметите никаких различий. Если нет (например, он может выполняться в node.js, Rhino и т. д.) или если вы не уверены, будьте осторожны.

Главные различия из области совместимости:

- В нестрогом режиме разрешены восьмеричные числовые литералы, например `0123` (83 в десятичной системе).
- Вызовы `window.escape(..)` и `window.unescape(..)` позволяют экранировать или отменять экранирование строк шестнадцатеричными последовательностями с ограничителями `%`. Пример: для `window.escape(" ?foo=97%&bar=3%")` генерируется строка `"%3Ffoo%3D97%25%26bar%3D3%25"`.
- Метод `String.prototype.substr` очень похож на `String.prototype.substring`, если не считать того, что вместо конечного индекса (без включения) во втором параметре передается длина (количество включаемых символов).

Web ECMAScript

В спецификации Web ECMAScript¹ рассматриваются различия между официальной спецификацией ECMAScript и текущими реализациями JavaScript в браузерах.

¹ <https://javascript.spec.whatwg.org/>.

Иначе говоря, эти элементы «обязательны» для браузеров (для обеспечения их совместимости друг с другом), но (на момент написания книги) они не перечислены в разделе «Дополнение В» официальной спецификации:

- <!-- и --> являются действительными ограничителями для однострочных комментариев.
- Дополнения `String.prototype` для возвращения строк в формате HTML: `anchor(..)`, `big(..)`, `blink(..)`, `bold(..)`, `fixed(..)`, `fontcolor(..)`, `fontsize(..)`, `italics(..)`, `link(..)`, `small(..)`, `strike(..)` и `sub(..)`.



Эти средства очень редко применяются на практике. Обычно вместо них используются другие встроенные DOM API или средства, определяемые пользователем.

- Расширения `RegExp`: `RegExp.$1 .. RegExp.$9` (группы совпадений) и `RegExp.lastMatch`/`RegExp["$&"]` (последнее совпадение).
- Расширения `Function.prototype`: `Function.prototype.arguments` (псевдоним для внутреннего объекта `arguments`) и `Function.caller` (псевдоним для внутреннего свойства `arguments.caller`).



`arguments` (а следовательно, и `arguments.caller`) считаются устаревшими, поэтому их следует по возможности избегать. Это вдвойне относится к псевдонимам, не используйте их!



Некоторые второстепенные и редко используемые отклонения не включены в список. За дополнительной информацией обращайтесь к внешним документам «Дополнение В» и «Web ECMAScript».

Все эти различия редко используются на практике, так что отклонения от спецификации не создают серьезных проблем. Просто будьте осторожны, если вы полагаетесь на них в своей работе.

Управляющие объекты

В четко сформулированных правилах поведения переменных в JS существуют исключения для переменных, которые определяются автоматически или иным образом создаются и передаются JS средой, используемой для размещения вашего кода (браузер и т. д.), — так называемые «управляющие объекты» (к числу которых относятся как встроенные объекты, так и функции).

Пример:

```
var a = document.createElement( "div" );  
  
typeof a;           // "object" – как  
                   // и ожидалось  
Object.prototype.toString.call( a ); // "[object  
                   // HTMLDivElement]"  
  
a.tagName;          // "DIV"
```

Здесь `a` не просто объект, а специальный управляющий объект, потому что он является элементом DOM. Он имеет особое внутреннее значение `[[Class]]` ("HTMLDivElement") и содержит заранее определенные (и часто неизменяемые) свойства.

Другая особенность таких объектов уже упоминалась в разделе «Ложные объекты» главы 4: некоторые объекты существуют, но при преобразовании в `boolean` они (как ни удивительно) дают `false` вместо ожидаемого значения `true`.

Другие особенности поведения управляющих объектов, о которых следует помнить:

- Недоступность нормальных встроенных средств `object` (таких, как `toString()`).
- Невозможность перезаписи.
- Некоторые предварительно определенные свойства, доступные только для чтения.
- Наличие методов, которые не могут `this`-переопределяться для других объектов.
- И так далее...

Управляющие объекты чрезвычайно важны для того, чтобы код JS мог работать в своем окружении. Но важно следить за тем, когда вы взаимодействуете с управляющим объектом, и не торопиться с предположениями относительно его поведения, так как оно часто не соответствует поведению обычных объектов JS.

Один важный пример управляющего объекта, с которым вы, вероятно, будете регулярно работать, – объект `console` и его различные функции (`log(..)`, `error(..)` и т. д.). Объект `console` предоставляется средой размещения специально для того, чтобы ваш код мог взаимодействовать с ним для выполнения различных операций вывода, связанных с разработкой.

В браузерах объект `console` связывается с консольным выводом средств разработчика, а в `node.js` и других серверных средах JS он обычно связывается с потоками стандартного вывода (`stdout`) и стандартных ошибок (`stderr`) системного процесса среды JavaScript.

Глобальные переменные DOM

Возможно, вы знаете, что объявление переменной в глобальной области видимости (с `var` или без) создает не только глобальную переменную, но и ее зеркальную копию: одноименное свойство в глобальном объекте (`window` в браузере).

Но менее известно то, что (из-за унаследованного поведения браузеров) при создании элементов DOM с атрибутами `id` создаются одноименные глобальные переменные. Пример:

```
<div id="foo"></div>
```

Также:

```
if (typeof foo == "undefined") {  
    foo = 42;           // Никогда не выполняется  
}  
  
console.log( foo ); // элемент HTML
```

Вероятно, вы привыкли управлять проверками глобальных переменных (с использованием `typeof` или `..` в проверках `window`) в предположении, что такие переменные создаются только кодом JS. Но как вы видите, они также могут создаваться контентом управляющей страницы HTML, что легко может привести вас в полное замешательство, если вы не будете осторожны.

И это еще одна причина, по которой вам следует (если это возможно) избегать глобальных переменных. А если без них не обойтись, используйте переменные с уникальными именами, у которых вероятность совпадения минимальна. Вы также должны следить за тем, чтобы избегать конфликтов имен не только с другим кодом, но и с контентом HTML.

Встроенные прототипы

Одна из самых известных и классических рекомендаций JavaScript: никогда не расширяйте встроенные прототипы.

Какое бы имя метода или свойства вы ни выбрали для добавления в `Array.prototype`, которое (еще) не существует, если это дополнение полезно, хорошо спроектировано и имеет подходя-

щее имя, скорее всего, оно *могло быть* добавлено в спецификацию, и в этом случае ваше расширение создаст конфликт.

Приведу реальный пример, который действительно случился со мной; он хорошо демонстрирует этот факт.

Я создавал встраиваемый виджет для других веб-сайтов, и мой виджет зависел от jQuery (хотя эта проблема могла возникнуть из-за любого другого фреймворка). Он работал почти на всех сайтах, но мы обнаружили один сайт, на котором он был полностью неработоспособен.

После почти недели анализа и отладки я обнаружил, что на этом сайте в одном из унаследованных файлов был глубоко запрятан код, который выглядел примерно так:

```
// В Netscape 4 нет Array.push
Array.prototype.push = function(item) {
    this[this.length-1] = item;
};
```

Если не считать безумного комментария (кого сейчас интересует Netscape 4?), выглядит разумно, правда?

Проблема в том, что уже после эпохи программирования для Netscape 4 в спецификацию был добавлен `Array.prototype.push`, но добавление было несовместимо с этим кодом. Стандартный вызов `push(..)` позволяет добавить сразу несколько элементов, а самодельная версия игнорировала все последующие элементы.

Практически все фреймворки JS содержат код, в котором `push(..)` используется с несколькими элементами. В моем случае это был код, связанный с селекторами CSS, и его работоспособность была полностью нарушена. Впрочем, аналогичные проблемы могли проявиться еще в десятках других мест.

Разработчик, который изначально написал эту «заплатку» для `push(..)`, инстинктивно присвоил ей имя `push`, но не предвидел

возможного занесения нескольких элементов. Безусловно, он действовал из лучших побуждений, но при этом заложил мину, которая сработала лишь почти через 10 лет, когда я неосознанно подключился к делу.

Из этой ситуации стоило бы вынести несколько уроков.

Во-первых, не расширяйте встроенные прототипы, если только вы твердо не уверены в том, что в этой среде будет выполняться только ваш код. Если вы не можете сказать этого со стопроцентной уверенностью, расширять встроенные прототипы опасно. Трезво оцените все риски.

Во-вторых, не определяйте расширения безусловно (потому что вы можете случайно перезаписать встроенные прототипы). В конкретном примере рассмотрим следующий код:

```
if (!Array.prototype.push) {  
    // В Netscape 4 нет Array.push  
    Array.prototype.push = function(item) {  
        this[this.length-1] = item;  
    };  
}
```

Здесь защитная команда `if` определяет самодельную версию `push()` только для тех сред JS, в которых она не существует. Вероятно, в моем случае это решило бы проблему. Тем не менее даже такой подход не защищен от риска:

1. Если бы код сайта (по какой-то невероятной причине) зависел от версии `push(..)`, которая игнорировала лишние элементы, работоспособность этого кода была бы нарушена много лет назад, когда появилась стандартная версия `push(..)`.
2. Если бы любая другая библиотека выдала собственную реализацию `push(..)` до защитной команды `if` и это было бы сделано несовместимым образом, это тоже привело бы к нарушению работы сайта.

Все это поднимает интересный вопрос, на который, честно говоря, разработчики JS не обращают должного внимания: стоит ли *когда-либо* полагаться на встроенное поведение, если ваш код выполняется в любой среде, где он не является единственным выполняемым кодом?

Строго говоря, на этот вопрос следовало бы ответить «нет», но это ужасно непрактично. Ваш код обычно не может переопределять собственные приватные версии всех встроенных аспектов поведения, от которых он зависит. Даже если бы это было возможно, такая тактика была бы слишком расточительной.

Итак, следует проводить функциональное тестирование встроенного поведения, а также тестирование на соответствие, чтобы убедиться в том, что код делает то, что ему положено? И что делать, если тесты не пройдут? Должен ли ваш код просто отказаться работать?

```
// не доверять Array.prototype.push
(function(){
    if (Array.prototype.push) {
        var a = [];
        a.push(1,2);
        if (a[0] === 1 && a[1] === 2) {
            // тесты прошли, использование безопасно!
            return;
        }
    }
    throw Error(
        "Array#push() is missing/broken!"
    );
})();
```

Теоретически выглядит здраво, но разрабатывать тесты для каждого встроенного метода тоже как-то непрактично.

Что же делать? «*Доверять, но проверять*» (проводить тестирование функциональности и соответствия) для всего? Или просто действовать по принципу «существует – значит, соответствует»

и дать возможность любым сбоям (происходящим из-за других) проявляться по мере их появления?

Идеального ответа нет. Единственное, что можно заметить, — все эти проблемы возникают только из-за расширения встроенных прототипов.

Если вы этого не делаете и никто другой не делает этого в коде вашего приложения — вы в безопасности. В противном случае следует заниматься разработкой по крайней мере с некоторой долей скептицизма, пессимизма и ожиданий возможной поломки.

Наличие полного набора модульных/регрессионных тестов вашего кода для всех известных сред — единственный способ выявить некоторые из этих проблем, но эти тесты никак не помогут защитить вас от этих конфликтов.

Прокладки совместимости (shims) полифили (polyfills)

Обычно говорят, что единственное безопасное место для расширения встроенных прототипов — старые (не соответствующие спецификации) среды, поскольку они вряд ли изменятся. Новые браузеры с новыми возможностями спецификации заменяют старые браузеры, а не расширяют их.

Если бы вы могли заглянуть в будущее и точно знать, как будут выглядеть завтрашние стандарты (например, для `Array.prototype.foobar`), то было бы полностью безопасно создать собственную совместимую версию для того, чтобы использовать ее сегодня, верно?

```
if (!Array.prototype.foobar) {  
    // глупо  
    Array.prototype.foobar = function() {  
        this.push( "foo", "bar" );  
    };  
}
```

Если для `Array.prototype.foobar` уже существует спецификация и поведение в ней эквивалентно этой логике, вы можете вполне безопасно определить такой фрагмент. В таком случае он обычно называется полифилом (или прокладкой совместимости).

Такой код очень полезно включать в вашу кодовую базу для старых браузерных сред, не обновляемых для новых спецификаций. Полифили — отличный способ создания кода с предсказуемым поведением для всех поддерживаемых сред.



ES5-Shim¹ — обширная подборка полифилов/прокладок, для того чтобы поднять проект до базовой линии ES5. Аналогичным образом ES6-Shim² предоставляет прокладки совместимости для новых API, добавленных в ES6. Хотя для API можно создавать прокладки совместимости/полифили, для нового синтаксиса это обычно невозможно. Чтобы преодолеть разрывы в синтаксисе, вам придется также использовать транспилятор ES6/ES5 (например, Traceur³).

Если ожидается появление стандарта и большинство обсуждений сходится на том, что как будет называться и как будет работать, создаваемый опережающий полифил для соответствия будущим стандартам называется «пролифил» (от «probably fill»).

Настоящие проблемы возникают тогда, когда для нового стандартного поведения невозможно (в полной мере) создать полизаполнение/пролизаполнение.

В сообществе идут обсуждения относительно того, допустимы ли частичные полифили (документирование частей, для которых

¹ <https://github.com/es-shims/es5-shim>.

² <https://github.com/es-shims/es6-shim>.

³ <https://github.com/google/traceur-compiler/wiki>.

невозможно создать полифил), или же полифилов следует избегать, если они не могут на сто процентов соответствовать спецификации.

Многие разработчики по крайней мере допускают некоторые распространенные частичные полифили (например, `Object.create(..)`), потому что части, для которых нет описаний, они использовать все равно не намеревались.

Некоторые разработчики полагают, что защитная команда `if` вокруг полифила/прокладки совместимости должна включать некую форму теста на соответствие, которая заменяет существующий метод, если он отсутствует или не проходит тест. По этой дополнительной прослойке тестирования соответствия прокладку совместимости (с тестированием соответствия) иногда отличают от полифила (проверка существования).

Единственный однозначный вывод заключается в том, что единственно *правильного* ответа не существует. Расширение встроенных прототипов, даже при «безопасном» исполнении в старых средах, не является абсолютно безопасным. То же самое можно сказать о зависимости от (возможно, расширенных) встроенных прототипов в присутствии чужого кода.

Оба подхода должны реализовываться с осторожностью, защитным программированием и обширным документированием возможных рисков.

<script>ы

Большинство сайтов/приложений, просматриваемых в браузере, содержат более одного файла с кодом. Очень часто в страницу включается несколько элементов `<script src=..></script>`, загружающих эти файлы по отдельности, и даже несколько элементов встроенного кода `<script> .. </script>`.

Но содержат ли отдельные файлы/фрагменты кода отдельные программы или же они в совокупности образуют одну программу JS?

Результат (возможно, неожиданный) заключается в том, что они действуют как независимые JS-программы в большинстве, хотя и не во всех отношениях.

Единственное, что у них есть *общего* – общий глобальный объект (`window` в браузере). Это означает, что несколько файлов могут присоединить свой код к общему пространству имен, и все они могут взаимодействовать друг с другом.

Таким образом, если один элемент `script` определяет глобальную функцию `foo()`, то при последующем выполнении второго элемента `script` он сможет обращаться к функции `foo()` и вызывать ее точно так же, как если бы он определил функцию сам.

Но *поднятие* видимости глобальных переменных (см. книгу «Область видимости и замыкания» этой серии) не переходит через эти границы, так что следующий код работать не будет (потому что объявление `foo()` еще не встречалось) независимо от того, используются ли встроенные элементы `<script> .. </script>` (как показано) или файлы `<script src=..></script>` с внешней загрузкой:

```
<script>foo();</script>
```

```
<script>
  function foo() { .. }
</script>
```

Но любой из этих двух фрагментов *будет* работать:

```
<script>
  foo();
  function foo() { .. }
</script>
```

Или:

```
<script>
  function foo() { .. }
</script>

<script>foo();</script>
```

Кроме того, если в элементе `script` (встроенным или внешнем) произойдет ошибка, как автономная отдельная JS-программа он остановится, но все последующие сценарии будут работать беспрепятственно (продолжая использовать общий глобальный объект).

Вы можете создать элементы `script` динамически в своем коде и встроить их в DOM страницы, и код будет работать фактические так же, как если бы он был нормально загружен в отдельном файле:

```
var greeting = "Hello World";

var el = document.createElement( "script" );

el.text = "function foo(){ alert( greeting );\
} setTimeout( foo, 1000 );";

document.body.appendChild( el );
```



Конечно, если в приведенном фрагменте задать в `el.src` URL-адрес некоторого файла (вместо того, чтобы присвоить `el.text` фактический код), вы фактически будете динамически создавать элемент `<script src src=..></script>` с внешней загрузкой.

Одно из различий между кодом во встроенным блоке и тем же кодом во внешнем файле заключается в том, что во встроенным блоке последовательность символов `</script>` встречаться не

может, так как (независимо от места размещения) она будет интерпретирована как конец программного блока. Итак, остерегайтесь следующего кода:

```
<script>
  var code = "<script>alert( 'Hello World' )</script>";
</script>
```

Выглядит безопасно, но `</script>` внутри строкового литерала приведет к аномальному завершению блока `script` и возникновению ошибки. Самое распространенное обходное решение:

```
"</sc" + "ript>";
```

Кроме того, учтите, что код во внешнем файле будет интерпретироваться в той кодировке (UTF-8, ISO-8859-8 и т. д.), в которой предоставляется файл (или в кодировке по умолчанию), но тот же код во встроенным элементе `script` в странице HTML будет интерпретироваться в кодировке страницы (или в кодировке по умолчанию).



Атрибут `charset` не будет работать со встроенными элементами `script`.

Другая устаревшая практика со встроенными элементами `script` — заключение встроенного кода в комментарии в стиле HTML или X(HT)ML:

```
<script>
<!--
alert( "Hello" );
//-->
</script>
<script>
<!--//--><![CDATA[//><!--
alert( "World" );
//--><!-->
</script>
```

Обе практики сейчас стали совершенно излишними. Если вы еще продолжаете применять их — прекращайте!



`<!--` и `-->` (комментарии в стиле HTML) в спецификации определяются как действительные односторонние комментарии-разделители (`var x = 2; <!--` действительный комментарий `-->` другой действительный комментарий) в JavaScript (см. раздел «Web ECMAScript») исключительно из-за этой устаревшей практики. Тем не менее никогда не используйте их.

Зарезервированные слова

В спецификации ES5, раздел 7.6.1, определяется набор «зарезервированных слов», которые не могут использоваться в качестве автономных имен переменных. С технической точки зрения они делятся на четыре категории: «ключевые слова», «слова, зарезервированные на будущее», литерал `null` и логические литералы `true`/`false`.

Ключевые слова, такие как `function` и `switch`, вполне очевидны. К категории слов, зарезервированных на будущее, относятся такие слова, как `enum`, хотя многие из них (`class`, `extends` и т. д.) сейчас используются ES6; также существуют другие зарезервированные слова только для режима `strict` (например, `interface`).

Пользователь StackOverflow «art4theSould» творчески переработал все эти зарезервированные слова в забавное маленькое стихотворение, которое можно найти по ссылке <https://stackoverflow.com/questions/26255/reserved-keywords-in-javascript/12114140#12114140>.

До появления ES5 зарезервированные слова также не могли быть именами свойств или ключей в объектных литералах, но это ограничение было снято.

Итак, следующая команда недопустима:

```
var import = "42";
```

А эта допустима:

```
var obj = { import: "42" };
console.log( obj.import );
```

Однако следует учитывать, что в некоторых старых версиях браузеров (прежде всего старые IE) эти правила применялись недостаточно последовательно. В некоторых местах использование зарезервированных свойств в именах свойств объектов может создать проблемы. Тщательно протестируйте все поддерживающие браузерные среды.

Ограничения реализации

Спецификация JavaScript не накладывает конкретные ограничения на такие аспекты, как количество аргументов у функций или длина строкового литерала. Такие ограничения существуют из-за особенностей реализации в разных движках.

Пример:

```
function addAll() {
    var sum = 0;
    for (var i=0; i < arguments.length; i++) {
        sum += arguments[i];
    }
    return sum;
}

var nums = [];
for (var i=1; i < 100000; i++) {
    nums.push(i);
}

addAll( 2, 4, 6 );           // 12
addAll.apply( null, nums ); // должно быть: 499950000
```

В некоторых движках JS вы получите правильный ответ 499950000, но в других (например, Safari 6.x) вы получите ошибку «`RangeError: превышение максимального размера стека вызовов`».

Другие известные примеры существующих ограничений:

- Максимальное количество символов, разрешенных в строковом литерале (не в значении `string`).
- Размер данных (в байтах), которые могут передаваться в аргументах при вызове функций.
- Количество параметров в объявлении функции.
- Максимальная глубина неоптимизированного стека вызовов (с рекурсией), то есть длина цепочки вызовов функций.
- Продолжительность непрерывного выполнения программы JS, блокирующей работу браузера.
- Максимально допустимая длина имени переменной.

Вам придется очень редко сталкиваться с этими ограничениями, но вы должны знать, что такие ограничения существуют, и что еще важнее — отличаются в зависимости от движка.

Итоги

Мы знаем и можем рассчитывать на тот факт, что сам язык JS имеет единый стандарт, который предсказуемо реализован всеми современными браузерами/движками. И это очень хорошо!

Однако JavaScript редко работает в изоляции. Он выполняется в среде, дополненной кодом сторонних библиотек, а иногда даже в ядрах/средах, отличных от браузерных сред.

Пристальное внимание к этим вопросам улучшает надежность вашего кода и делает его более понятным.

Об авторе

Кайл Симпсон – евангелист Open Web из Остина (штат Техас), большой энтузиаст всего, что касается JavaScript. Автор нескольких книг, преподаватель, спикер и участник/лидер проектов с открытым кодом.

Кайл Симпсон

{Вы не знаете JS}

Типы и грамматические конструкции

Перевел с английского Е. Матвеев

Заведующая редакцией

Ю. Сергиенко

Ведущий редактор

К. Тульцева

Литературный редактор

А. Бульченко

Художественный редактор

В. Мостипан

Корректоры

С. Беляева, М. Молчанова

Верстка

Л. Егорова

Изготовлено в России. Изготовитель: ООО «Прогресс книга».

Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,

Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 05.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12.000 —

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 13.05.19. Формат 60x90/16. Бумага офсетная. Усл. п. л. 15,000.

Тираж 1000 экз. Заказ Р-947.

Отпечатано в типографии ООО «ИНФО СИСТЕМ».

420044, г. Казань, пр. Ямашева, д. 36Б.

ИЗДАТЕЛЬСКИЙ ДОМ «ПИТЕР» предлагает
профессиональную, популярную и детскую развивающую литературу

Заказать книги оптом можно в наших представительствах

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-83, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электроразводская», Семеновская наб., д. 2/1, стр. 1, 6 этаж
тел./факс: (495) 234-38-15; e-mail: sales@msk.piter.com

Воронеж: тел.: 8 951 861-72-70; e-mail: hitsenko@piter.com

Екатеринбург: ул. Толедова, д. 43а; тел./факс: (343) 378-98-41, 378-98-42;
e-mail: office@ekat.piter.com; skype: ekat.manager2

Нижний Новгород: тел.: 8 930 712-75-13; e-mail: yashny@yandex.ru; skype: yashny1

Ростов-на-Дону: ул. Ульяновская, д. 26
тел./факс: (863) 269-91-22, 269-91-30; e-mail: piter-ug@rostov.piter.com

Самара: ул. Молодогвардейская, д. 33а, офис 223
тел./факс: (846) 277-89-79, 277-89-66; e-mail: pitvolga@mail.ru,
pitvolga@samara-ttk.ru

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163; тел./факс: +37 517 208-80-01, 208-81-25;
e-mail: og@minsk.piter.com

Издательский дом «Питер» приглашает к сотрудничеству авторов:
тел./факс: (812) 703-73-72, (495) 234-38-15; e-mail: ivanova@piter.com
ПогроБная информация здесь: <http://www.piter.com/page/avtoru>

**Издательский дом «Питер» приглашает к сотрудничеству зарубежных
торговых партнеров или посредников, имеющих выход на зарубежный
рынок:** тел./факс: (812) 703-73-73; e-mail: sales@piter.com

Заказ книг для вузов и библиотек:
тел./факс: (812) 703-73-73, доб. 6243; e-mail: uchebnik@piter.com

Заказ книг по почте: на сайте www.piter.com; тел.: (812) 703-73-74, доб. 6216;
e-mail: books@piter.com

Вопросы по продаже электронных книг: тел.: (812) 703-73-74, доб. 6217;
e-mail: kuznetsov@piter.com

КНИГА-ПОЧТОЙ



ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: www.piter.com
- по электронной почте: books@piter.com
- по телефону: (812) 703-73-74

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщают по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы, (адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**

ВАША УНИКАЛЬНАЯ КНИГА

Хотите издать свою книгу? Она станет идеальным подарком для партнеров и друзей, отличным инструментом для продвижения вашего бренда, презентом для памятных событий! Мы сможем осуществить ваши любые, даже самые смелые и сложные, идеи и проекты.

Мы предлагаем:

- издать вашу книгу
- издание книги для использования в маркетинговых активностях
- книги как корпоративные подарки
- рекламу в книгах
- издание корпоративной библиотеки

Почему надо выбрать именно нас:

Издательству «Питер» более 20 лет. Наш опыт – гарантия высокого качества.

Мы предлагаем:

- услуги по обработке и доработке вашего текста
- современный дизайн от профессионалов
- высокий уровень полиграфического исполнения
- продажу вашей книги во всех книжных магазинах страны

Обеспечим продвижение вашей книги:

- рекламой в профильных СМИ и местах продаж
- рецензиями в ведущих книжных изданиях
- интернет-поддержкой рекламной кампании

Мы имеем собственную сеть дистрибуции по всей России, а также на Украине и в Беларуси. Сотрудничаем с крупнейшими книжными магазинами.

Издательство «Питер» является постоянным участником многих конференций и семинаров, которые предоставляют широкую возможность реализации книг.

Мы обязательно проследим, чтобы ваша книга постоянно имелась в наличии в магазинах и была выложена на самых видных местах.

Обеспечим индивидуальный подход к каждому клиенту, эксклюзивный дизайн, любой тираж.

Кроме того, предлагаем вам выпустить электронную книгу. Мы разместим ее в крупнейших интернет-магазинах. Книга будет сверстана в формате ePUB или PDF – самых популярных и надежных форматах на сегодняшний день.

Свяжитесь с нами прямо сейчас:

Санкт-Петербург – Анна Титова, (812) 703-73-73, titova@piter.com
Москва – Сергей Клебанов, (495) 234-38-15, klebanov@piter.com



«Отличная книга для понимания неочевидных концепций JavaScript, которые облегчат жизнь любого разработчика».

— Отзыв читателя с Amazon

ТИПЫ & ГРАММАТИЧЕСКИЕ КОНСТРУКЦИИ

Каким бы опытом программирования на JavaScript вы ни обладали, скорее всего, вы не понимаете язык в полной мере. Это лаконичное руководство исследует типы более глубоко, чем все существующие книги: вы узнаете, как работают типы, о проблемах их преобразования и научитесь пользоваться новыми возможностями.

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

В КНИГЕ РАССМАТРИВАЮТСЯ СЛЕДУЮЩИЕ ТЕМЫ:

- Семь типов данных JavaScript: null, undefined, boolean, number, string, object и symbol.
- Уникальные особенности массивов, строк и чисел JavaScript, их достоинства и недостатки.
- Встроенные объектные обертки для примитивных значений.
- Странности преобразования типов и его полезность во многих ситуациях.
- Различные нюансы синтаксиса JavaScript: команды, выражения и т. д.

КАЙЛ СИМПСОН — евангелист Open Web и энтузиаст JavaScript. Он пишет книги, читает лекции, выступает на семинарах, конференциях и является лидером OSS.

ISBN: 978-5-4461-1266-1



9 785446 112661



Заказ книг:

тел.: (812) 703-73-74
books@piter.com

instagram.com/piterbooks

youtube.com/ThePiterBooks

vk.com/piterbooks

facebook.com/piterbooks

WWW.PITER.COM
каталог книг и интернет-магазин