Используй лучшее в JavaScript

JavaScript
сильные стороны

Дуглас Крокфорд

O'REILLY® YAHOO! PRESS ПИТЕР®

Douglas Crockford

JavaScript
The Good Parts

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

Дуглас Крокфорд

JavaScript
сильные стороны

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2012

ББК 32.988.02-018.1
УДК 004.43
К83

Крокфорд Д.
К83 JavaScript: сильные стороны. — СПб.: Питер, 2012. — 176 с.: ил.

ISBN 978-5-459-01263-7

Любой язык программирования имеет свои сильные и слабые стороны, однако язык JavaScript в большей степени состоит из последних, так как был разработан в спешке и практически не отлаживался. В этой книге среди множества самых ужасных JavaScript-конструкций выделены наиболее надежные, понятные и удобные в сопровождении — то подмножество языка, которое позволяет создавать по-настоящему расширяемый и эффективный код.

Автор — уважаемый среди разработчиков эксперт в области JavaScript — выделяет массу интересных идей, которые делают JavaScript прекрасным объектно-ориентированным языком программирования. Эта книга откроет вам красивый, элегантный, простой и выразительный язык программирования, позволяющий создавать эффективный код независимо от того, управляете вы библиотеками объектов или просто пытаетесь заставить Ajax работать быстрее. Если вы разрабатываете сайты или приложения для Интернета, эта книга вам просто необходима.

ББК 32.988.02-018.1
УДК 004.43

ISBN 978-0596517748 англ.

ISBN 978-5-459-01263-7

Краткое оглавление

Оглавление

Об авторе

Дуглас Крокфорд — ведущий специалист по JavaScript в Yahoo! Он известен как создатель и популяризатор формата JSON (JavaScript Object Notation). Он наиболее авторитетный из ныне живущих специалистов по JavaScript. Он регулярно принимает участие в конференциях, посвященных передовым веб-технологиям, а также является членом комитета ECMAScript.

Моим друзьям: *Клементу*, *Филберту*, *Сеймуру*, *Стерну и, чтобы не забыть, К. Твилдо*

Предисловие

Коль не удастся нам пиеса, мы желали,
Чтоб знали вы, что мы не с тем пришли сюда,
Чтоб нам не удалось; мы вот чего искали:
Вам предложить свои услуги, господа;
Вот нашего конца вернейшее начало.

Уильям Шекспир. Сон в летнюю ночь

Эта книга по программированию на JavaScript. Она предназначена для тех, кто случайно или из любопытства впервые решил познакомиться с JavaScript, а также для тех, кто уже немного работал с этим языком и готов поднять свои отношения с ним на новый уровень. JavaScript — удивительно мощный язык. И хотя изучение JavaScript представляется затруднительным в силу его нетрадиционности, это компенсируется его небольшим объемом.

Моя цель — помочь вам научиться думать в терминах JavaScript. Я покажу вам некоторые компоненты языка и помогу найти пути, которые позволят объединить их в завершенных конструкциях. Эта книга — не справочник. Представленная здесь информация о языке JavaScript и его особенностях не является исчерпывающей. Все нюансы вы можете легко найти в Интернете. Эта книга содержит только то, что действительно важно.

Эта книга не для начинающих. Когда-нибудь я надеюсь написать книгу «JavaScript: первые шаги», но это не она. Это книга не о библиотеке Ajax и не о веб-программировании. Основное внимание в ней сконцентрировано на языке JavaScript, являющимся лишь одним из языков, которыми должен овладеть веб-разработчик.

Эта книга не для чайников. Она небольшая, но содержательная. В ней собрано много материала. Не расстраивайтесь, если придется прочесть ее не один раз. Ваши старания будут вознаграждены.

Условные обозначения

В этой книге используются следующие условные обозначения:

- *Курсивом* обозначены термины, встречающиеся в книге впервые.
- Специальным шрифтом выделены URL-адреса, а также названия и расширения файлов.
- `Моноширинным шрифтом` обозначены фрагменты компьютерного кода в широком смысле, включая команды, опции, переменные, атрибуты, ключи, запросы, функции, методы, типы, классы, модули, свойства, параметры, значения, объекты, события, обработчики событий, XML- и XHTML-теги, макросы и ключевые слова.

Использование примеров кода

Эта книга написана, чтобы помочь вам справиться с поставленной задачей. В общем, вы можете использовать код, приведенный в этой книге, в своих программах и документации. Для этого вам не нужно разрешение. Так, к примеру, чтобы написать программу, содержащую несколько фрагментов кода из этой книги, разрешение не потребуется. А вот продажа или распространение на компакт-дисках примеров уже требует разрешения. Разрешение не нужно, если вы, отвечая на вопрос, процитировали эту книгу и привели несколько примеров кода. В то же время включение значительного количества примеров кода из этой книги в документацию по вашему продукту уже требует разрешения.

Если вам кажется, что использование примеров кода выходит за рамки упомянутых здесь требований, не стесняйтесь обращаться к нам по адресу permissions@oreilly.com.

Благодарности

Я хочу поблагодарить экспертов, отыскавших множество моих вопиющих ошибок. Что может быть лучше, чем ситуация, когда действительно умные люди замечают ваши промахи? Но еще лучше, если это происходит, прежде чем ваши ошибки станут достоянием общественности. Спасибо Стиву Соудерсу (Steve Souders), Биллу Скотту (Bill Scott), Жульен Леконта (Julien Lecomte), Стояну Стефанову (Stoyan Stefanov), Эрику Мираджила (Eric Miraglia) и Эллиотт Расти Гарольд (Elliotte Rusty Harold).

Я хочу поблагодарить людей, с которыми работал в Electric Communities и State Software, ведь они помогли мне обнаружить столько хорошего в JavaScript. Особая

благодарность Чипу Морнингстару (Chip Morningstar), Ренди Фармеру (Randy Farmer), Джону Ла Марк Миллеру (John La, Mark Miller), Скотту Шаттаку (Scott Shattuck) и Биллу Эдни (Bill Edney).

Я хочу поблагодарить компанию Yahoo! за предоставленное мне время для работы над этим проектом и за прекрасное рабочее место. Спасибо всем бывшим и нынешним членам Ajax Strike Force. Спасибо O'Reilly Media, Inc., в частности Мари Треслер (Mary Treseler), Саймону Сен Лорану (Simon St.Laurent) и Самиту Мукерджи (Sumita Mukherji), благодаря которым все прошло так гладко.

Особая благодарность профессору Лизе Дрейк (Lisa Drake) за все то, что она делает. Кроме того, я хочу поблагодарить ребят из ECMA TC39, упорно пытающихся улучшить ECMAScript.

Наконец, спасибо Брендану Эйчи (Brendan Eich), автору наиболее недооцененного языка программирования в мире, без которого эта книга была бы не нужна.

От издательства

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

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на сайте издательства http://www.piter.com.

1 Сильные стороны

...привлекательная внешность, мужественная осанка — вот и все мое колдовство.

Уильям Шекспир. Виндзорские насмешницы

В начале своей карьеры программиста я хотел досконально изучить языки, на которых писал, чтобы в полной мере использовать все их возможности. Думаю, это был отличный способ проявить себя, и это сработало, я стал тем, к кому можно обратиться, чтобы узнать, как использовать ту или иную функцию.

Со временем я понял, что использование некоторых функций приносит больше вреда, чем пользы. Одни функции были плохо определены, что, вероятнее всего, объяснялось проблемами переносимости. Другие превращали код в нечитабельный и плохо модифицируемый, что делало его чрезмерно сложным и было чревато ошибками. Кроме того, некоторые функции были плохо спроектированы. Иногда разработчики языка тоже совершают ошибки.

Любой язык программирования имеет свои сильные и слабые стороны. Я понял, что мог бы стать хорошим программистом, используя только достоинства и избегая недостатков. В конце концов, как получить что-то хорошее из плохих деталей?

Иногда комитеты по стандартизации могут убрать из языка неудачные фрагменты, способные привести к сбою программ, в которых эти несовершенные фрагменты используются. Но, как правило, они лишь добавляют новые функции к многочисленным уже существующим и не слишком удачным, причем эти новые функции не всегда гармонично сочетаются с существующими, что порождает новые проблемы.

Тем не менее у *вас* есть возможность определять собственные подмножества и создавать хорошие программы, основываясь исключительно на удачных фрагментах.

Язык JavaScript почти полностью состоит из недостатков. Все началось с того, что основы были заложены в небывало короткий промежуток времени. У раз-

работчиков так и не нашлось времени, чтобы все опробовать и отладить. То же произошло и с Netscape Navigator 2 — все было сделано довольно грубо. Идея создания Java™-приложений провалилась, и язык JavaScript занял место «основного языка для Веб». Хотя популярность JavaScript почти не зависит от его качеств как языка программирования.

К счастью, у JavaScript есть и довольно сильные стороны. Это красивый, элегантный и очень выразительный язык, похороненный под грудой добрых намерений и ошибок. Лучшие черты JavaScript так старательно скрыты, что на протяжении многих лет бытует мнение о том, что JavaScript — это просто невзрачная сломанная игрушка. Моя цель — открыть сильные стороны JavaScript, как выдающегося из динамических языков программирования. Будем считать, что JavaScript — это кусок мрамора, и я, отсекая все лишнее, попытаюсь показать его истинную суть. Я считаю, что мне удалось выделить более элегантное подмножество, чем JavaScript в целом, более надежное, удобное для чтения и сопровождения.

Эта книга не является детальным описанием JavaScript. Основное внимание здесь сосредоточено на сильных сторонах этого языка и советах, помогающих избежать слабых. Упомянутые в этой книге конструкции могут быть использованы для построения надежных и понятных программ любого объема. Концентрируясь только на хороших конструкциях, можно сократить время обучения, повысить надежность и спасти несколько деревьев (которые иначе пришлось бы извести на лишнюю бумагу).

Возможно, наибольшая польза от изучения сильных сторон языка состоит в том, что это не требует отучиваться от использования его слабых сторон. Отучиться применять плохие программные шаблоны очень трудно. Большинство из нас сталкивается с этой непростой задачей с крайней неохотой. Иногда языки подразделяют на подмножества, чтобы облегчить работу студентов. Однако в данном случае я выделил такие подмножества JavaScript, которые способны облегчить работу профессионалам.

Почему JavaScript?

JavaScript важен, потому что это язык веб-браузера. Его связь с браузером делает JavaScript одним из самых популярных языков программирования в мире, хотя в то же время он один из самых презираемых. API браузера, объектная модель документов (Document Object Model, DOM) просто ужасны, но обвинять JavaScript несправедливо. На любом языке нелегко работать с DOM, так как эта модель плохо определена и непоследовательно реализована. Эта книга вскользь касается DOM. Я думаю, что написать книгу о сильных сторонах DOM было бы крайне сложно.

JavaScript презирают потому, что этот язык не похож на какой-либо другой. Если вы разбираетесь в каком-нибудь другом языке, но вам приходится рабо-

тать в программной среде, которая поддерживает только JavaScript, это раздражает. Большинство людей в такой ситуации даже не стараются вначале изучить JavaScript, а потом удивляются, когда оказывается, что JavaScript отличается от других языков, которые они предпочли бы использовать, и что эти различия существенны.

У JavaScript есть удивительная особенность — этот язык позволяет решать поставленные задачи, даже не имея особого представления ни о самом языке, ни о программировании вообще. Он обладает огромной выразительной силой. Тем не менее лучше, если вы знаете, что делаете. Программирование — дело трудное. И без должных знаний браться за него не стоит.

Анализ JavaScript

В основу JavaScript положены как очень хорошие, так и очень плохие идеи.

Очень хорошие идеи касаются функций, нестрогой типизации, динамических объектов, а также нотации литералов объектов. Плохие же идеи связаны с моделью программирования на основе глобальных переменных.

Функции в JavaScript представляют собой различного рода объекты (в основном) в лексическом контексте. JavaScript — один из основных языков, использующих лямбда-выражения. По сути, JavaScript имеет гораздо больше общего с Lisp и Scheme, нежели чем с Java. JavaScript — это Lisp в «шкуре» C, что делает его удивительно мощным языком.

Сегодня в большинстве языков программирования мода требует строгой типизации. Существует теория, что строгая типизация позволяет компилятору обнаруживать больше ошибок во время компиляции. Чем скорее мы сможем найти и исправить ошибки, тем дешевле они нам обойдутся. В JavaScript нет строгой типизации, поэтому компиляторы не в состоянии обнаружить ошибки типизации. Это вызывает беспокойство людей, которые пришли в JavaScript из языков со строгой типизацией. Однако оказывается, что строгая типизация не избавляет от необходимости тщательного тестирования. Я обнаружил, что некоторые из ошибок, которые компилятор определяет как ошибки типизации, вовсе не являются ошибками, и это настораживает. Я вообще считаю, что решать подобные проблемы поможет нестрогая типизация. Совершенно не обязательно выстраивать сложные иерархии классов и ломать голову над системой типов, чтобы достигнуть желаемого результата.

В JavaScript реализована очень мощная нотация литералов объектов. Объекты создаются простым указанием их компонентов. Эта нотация привела к появлению популярного формата обмена данными JSON. (Подробнее о JSON рассказано в приложении Д.)

Прототипическое наследование в JavaScript — спорная концепция. Объекты в JavaScript не привязаны к классам и наследуют свойства непосредственно от

других объектов. На самом деле, это довольно мощный инструмент, но он не знаком программистам с классическим образованием. Если вы попытаетесь применять классические шаблоны разработки непосредственно в JavaScript, вы будете разочарованы. Однако если вы научитесь работать с прототипами в контексте JavaScript, ваши усилия будут вознаграждены.

Язык JavaScript сильно пострадал из-за клеветы, касающийся его основных идей. По большей части, эти идеи хороши, если не необычны. Но кое-что действительно ужасно: JavaScript зависит от глобальных переменных. Все переменные высшего уровня собраны вместе в общем пространстве имен, названном глобальным объектом. Это плохо, так как использование глобальных переменных — не лучшее решение, но в JavaScript они являются основополагающими. К счастью, в JavaScript предусмотрены инструменты, способные решить эту проблему.

Некоторые недостатки невозможно игнорировать. Иногда они неизбежно приводят к ужасным последствиям, о чем рассказывается в приложении А. Однако избежать большинства из них можно, обобщая написанное в приложении Б. Обратитесь к любой другой книге по JavaScript, если захотите побольше узнать о недостатках.

Стандарт, определяющий JavaScript (известный как JScript) — третье издание языка ECMAScript, доступен по адресу http://www.ecma-international.org/publications/files/ecma-st/ECMA-262.pdf. В этой книге описывается не весь язык, обладающий массой недостатков, а лишь подмножество ECMAScript. Приводимые рекомендации не являются исчерпывающими, но помогают в крайних случаях, которых следует избегать.

В приложении В описан программный продукт под названием JSLint. Это анализатор JavaScript, который проверяет код, написанный на JavaScript, и сообщает о его недостатках. JSLint обеспечивает точность, которой не хватает для развития JavaScript, и дает вам уверенность в том, что ваши программы состоят только из хороших частей.

JavaScript — это язык контрастов. В нем много ошибок и сложностей, которые заставляют задаться вопросом: «Почему я должен использовать JavaScript?» Есть два ответа. Во-первых, у вас всегда есть выбор. Сеть стала важной площадкой для разработки приложений, а JavaScript — единственный язык, который поддерживается всеми браузерами. Очень жаль, что идея Java провалилась, она могла бы стать альтернативой для тех, кому нужен строго типизированный классический язык. Но Java потерпел неудачу, а JavaScript процветает, так что еще не известно, так ли плох JavaScript.

Во-вторых, несмотря на все свои недостатки, JavaScript действительно хорош. Это легкий и выразительный язык. И приобретя некоторые навыки, вы поймете, что функциональное программирование — дело довольно веселое.

Однако для того чтобы использовать язык хорошо, нужно быть хорошо информированным об ограничениях, которые порой ставят в тупик. Но пусть это вам не мешает. Сильные стороны с лихвой компенсируют слабые.

Простая проверка

Веб-браузер и любой текстовый редактор — это все, что необходимо для запуска программ на JavaScript. Во-первых, создайте HTML-файл и назовите его, к примеру, program.html:

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

Затем создайте в том же каталоге файл с именем program.js:

```
document.writeln('Hello, world!');
```

Теперь, чтобы увидеть результат, откройте ваш HTML-файл в браузере. На протяжении всей книги метод `method` используется для определения новых методов. Он определяется следующим образом:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

Подробнее этот метод описан в главе 4.

2 Грамматика

... знаком он мне: В грамматике
читал его когда-то.

Уильям Шекспир. Тит Андроник

Описанная в этой главе грамматика сильных сторон JavaScript дает общее представление о структуре языка. Описать грамматику можно с помощью синтаксических диаграмм.

Правила интерпретации таких диаграмм довольно просты:

- Синтаксические диаграммы читаются слева направо.
- Овалами обозначены символьные значения, прямоугольниками — правила или описания.
- Верная последовательность может быть получена из элементов, стоящих друг за другом.
- Любая другая последовательность не верна.
- Синтаксическая диаграмма, имеющая на концах одинарные штрихи, может быть вставлена между двумя другими диаграммами, но с диаграммами, имеющими на концах двойные штрихи, это невозможно.

Грамматика сильных сторон JavaScript, описанная в этой главе, значительно проще грамматики всего языка.

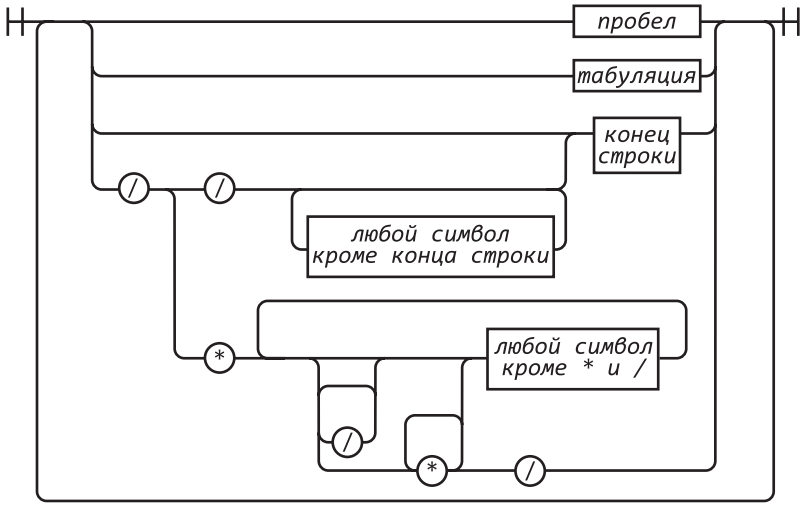
Пробельные символы

Пробелы применяются для разделения символов или комментариев. Пробельные символы, как правило, не важны, но иногда их приходится использовать для разделения последовательностей символов, которые могли бы быть объединены. Например:

```
var that = this;
```

Пробел между `var` и `that` удалять не следует, хотя другие пробелы могут быть удалены.

пробел



В JavaScript существует два способа задания комментариев: блочные комментарии начинаются с `/*` и заканчиваются `*/`, строчный комментарий начинается с `//`. Чтобы облегчить чтение кода, принято использовать комментарии, точно описывающие программный код. Нет ничего хуже, чем устаревшие комментарии.

Блочное задание комментариев символами `/* */` пришло из языка PL/1. В PL/1 такое сочетание было выбрано потому, что оно вряд ли могло бы встретиться в программе, разве что внутри строки. В JavaScript эти сочетания могут возникнуть и в регулярных выражениях, поэтому блочные комментарии не совсем безопасны. Например:

```
/*
    var rm_a = /a*/.match(s);
*/
```

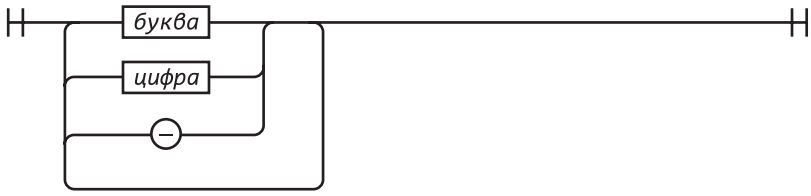
Приведенный код содержит ошибку, поэтому рекомендуется избегать символов `/* */` в комментариях, а использовать для комментариев символы `//`. В этой книге применяется исключительно второй способ.

Имена

Имя может состоять из одной и более букв, цифр или символов подчеркивания. В качестве имени нельзя использовать одно из следующих зарезервированных слов:

```
abstract
boolean break byte
case catch char class const continue
debugger default delete do double
else enum export extends
false final finally float for function
goto
if implements import in instanceof int interface
long
native new null
package private protected public
return
short static super switch synchronized
this throw throws transient true try typeof
var volatile void
while with
```

название

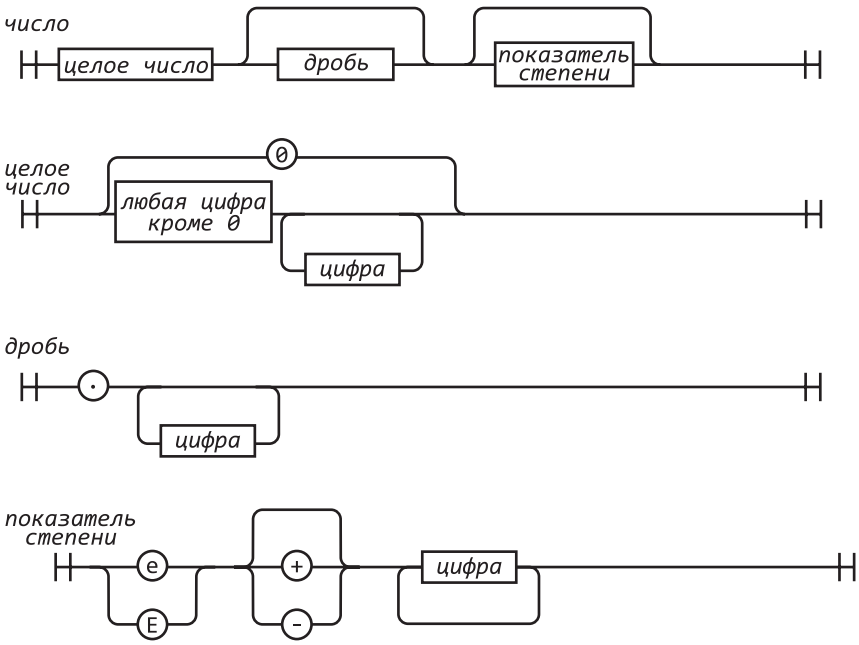


Большинство из этих зарезервированных слов в JavaScript не используются. При этом в приведенный список не включены несколько слов, которые следует зарезервировать, например `undefined`, `NaN` и `Infinity`. Не допускается использование зарезервированных слов для именования переменных или параметров. Более того, их нельзя применять в качестве имен параметров объекта, причем ни при символьном задании объекта, ни после точки.

Имена служат для объявления переменных, параметров, свойств, операторов и меток.

Числа

В JavaScript всего один числовой тип. Как и `double` в Java, он представляет собой 64-разрядное число с плавающей точкой. В отличие от большинства других языков программирования, в JavaScript нет отдельного целого типа, поэтому 1 и 1.0 — это одно и то же значение. Это довольно удобно, так как полностью исключает проблему переполнения для небольших целых чисел. Все, что вам нужно знать о числе — это то, что оно действительно число. Таким образом, удается избежать множества ошибок, связанных с числовыми типами.



Если числовое значение содержит показатель степени, то оно буквально вычисляется путем умножения на 10 части, стоящей перед `e`, столько раз, сколько указано после `e`. Так, 100 и 1e2 — это одно и то же.

Для задания отрицательных чисел используется префикс `-`.

Для обозначения результатов вычислений, в ходе которых не может быть получен нормальный результат, используется значение `NaN`. `NaN` не имеет конкретного численного значения. Функция `isNaN(number)` проверяет, имеет ли число значение `NaN`.

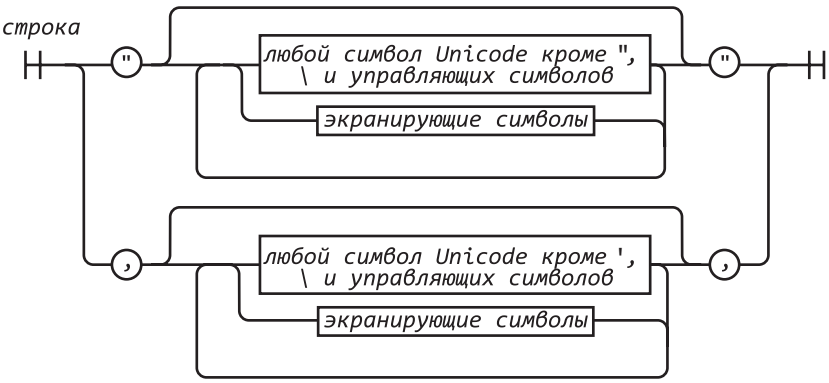
Бесконечностью считаются все значения больше 1.79769313486231570e +308.

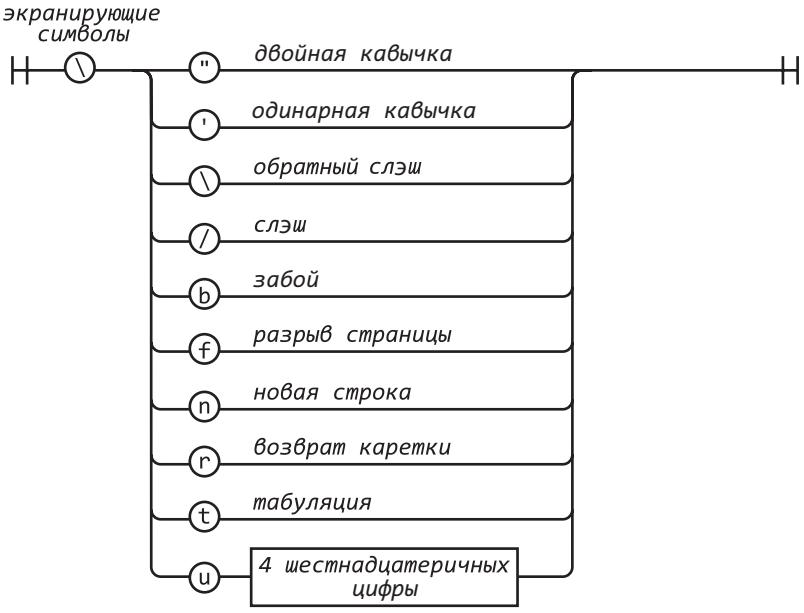
К числам применяются методы, которые подробно описаны в главе 8. В JavaScript существует объект `Math`, содержащий набор методов для работы с числами. Например, метод `Math.floor(number)` преобразует значения чисел в целые.

Строки

Строка может быть заключена в одинарные или двойные кавычки. Она может содержать ноль и более символов. Обратный слэш (\) является экранирующим символом. В то время когда разрабатывался язык JavaScript, стандарт Unicode состоял из набора 16-разрядных символов, так что все символы в JavaScript являются 16-разрядными.

JavaScript не имеет символьного типа. Для того чтобы получить символ, нужно задать строку из одного символа.





Управляющие последовательности позволяют вставлять в строки символы, которые обычно вставлять не разрешается, например обратный слэш, кавычки и управляющие символы. Последовательность `\u` позволяет задать кодовое значение символа.

```
"" === "\ U0041"
```

У строк есть свойство `length`. Например, `"seven".length` равно 5.

Строки являются постоянными. После того как строка создана, она не может быть изменена. Однако можно легко создать новую строку, объединив другие строки вместе с помощью оператора `+`.

Две строки, содержащие одни и те же символы в одном и том же порядке, считаются одинаковыми. Например:

```
'c' + 'a' + 't' === 'cat'
```

Это равенство верно.

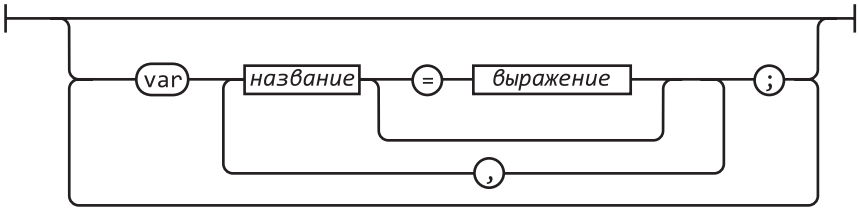
У строк имеются методы (см. главу 8), например:

```
'cat'.toUpperCase() === 'CAT'
```

Инструкции

Компилируемый модуль содержит набор исполняемых инструкций. Каждый тег `<script>` указывает веб-браузеру фрагмент, который необходимо скомпилировать и немедленно выполнить. Не имея компоновщика, JavaScript собирает их все вместе в общем глобальном пространстве имен. Подробное описание глобальных переменных приведено в приложении A.

объявление переменных



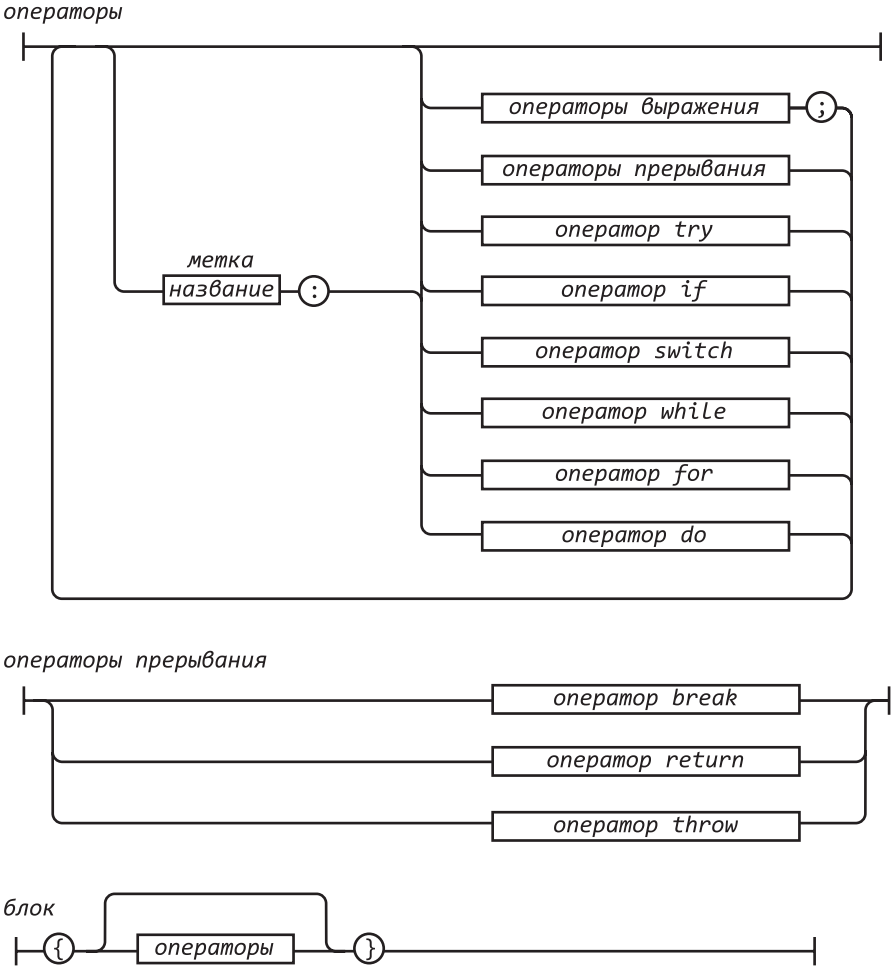
Для объявления закрытых переменных инструкция `var` помещается внутрь функции.

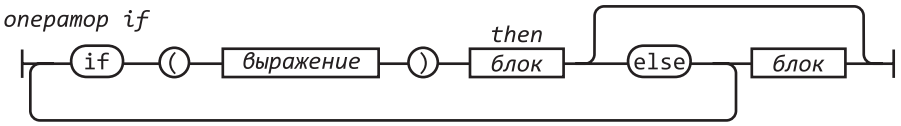
Инструкции `switch`, `while`, `for` и `do` могут иметь дополнительный префикс `label`, который используется с инструкцией `break`.

Как правило, инструкции выполняются по порядку сверху вниз. Последовательность выполнения может быть изменена с помощью условных операторов (`if` и `switch`), циклов (`while`, `for` и `do`), прерываний (`break`, `return` и `throw`) и вызовов функций.

Блок представляет собой набор инструкций, заключенных в фигурные скобки. В отличие от многих других языков, блоки в JavaScript не создают новой области видимости, так что определять переменные следует в начале функции, а не в блоках.

Инструкция `if` изменяет ход выполнения программы в зависимости от значения выражения. Если значение выражения *истинно*, выполняется блок, следующий за `then`, в противном случае будет выбрана ветвь, следующая за `else`.

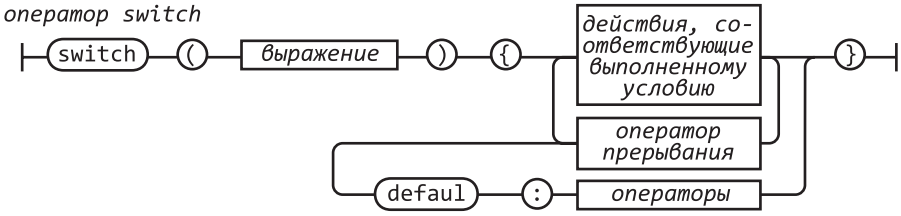




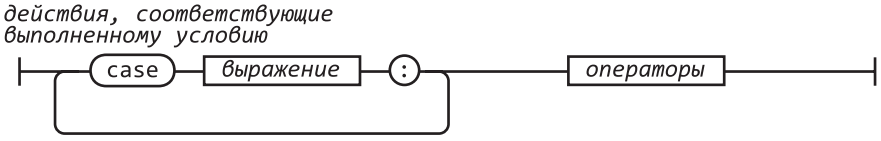
Ложными являются следующие значения:

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

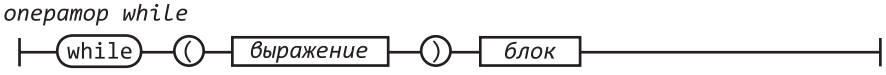
Все другие значения являются истинными, в том числе строка `'false'` и все объекты.



Оператор `switch` выполняет множество ветвей. Он проверяет равенство выражения всем ветвям `case`. Выражение может быть числом или строкой. Если найдено точное соответствие, выполняются инструкции соответствующей ветви. Если совпадения нет, выполняются инструкции ветви `default`.



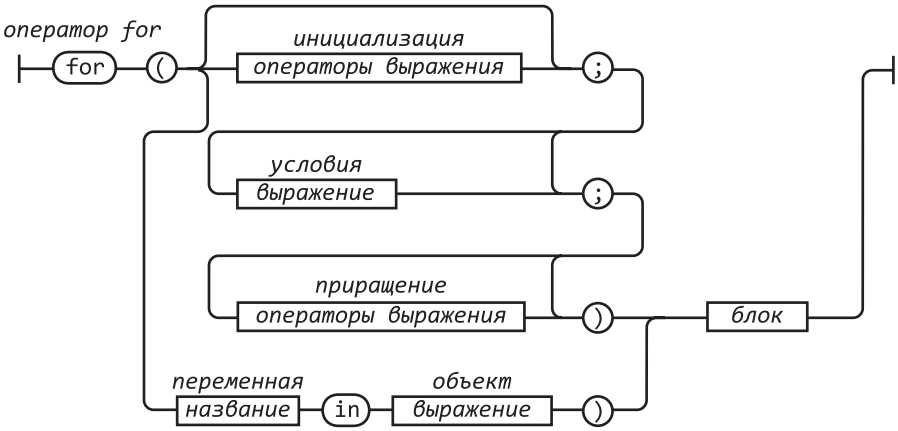
Предложение `case` содержит одно или несколько выражений. Выражения для сравнения не должны быть постоянными. Каждый блок внутри `case` должен содержать инструкцию прерывания, чтобы предотвратить переход к следующему условию. Для выхода из инструкции `switch` необходимо использовать инструкцию `break`.



С помощью инструкции `while` можно организовать цикл. Если выражение становится ложным, то выполнение цикла перерывается. Пока выражение истинно, блок выполняется.

Используя инструкцию `for`, можно организовать более сложный цикл. Существует две формы цикла `for`.

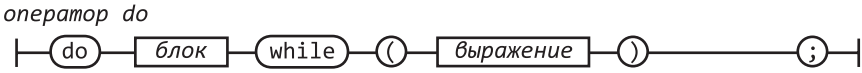
Если цикл задан в обычной форме, то работа с ним осуществляется в три этапа: инициализация, проверка условия, приращение. Во-первых, выполняется инициализация, которая обычно задает начальные переменные цикла. Затем оценивается состояние. Как правило, переменная цикла проверяется на соответствие критерию завершения цикла. Если условие опущено, предполагается, что условие истинно. Если условие ложно, цикл прерывается. В противном случае при исполнении блока выполняется приращение, а затем цикл повторяется с проверки условия.



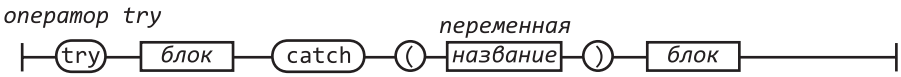
В другой форме (имеется в виду `for in`) перечисляют имена свойств (или ключей) объекта. На каждой итерации переменной присваивается очередная строка, соответствующая имени свойства объекта.

Обычно для проверки используется метод `object.hasOwnProperty(variable)`, определяющий, обладает ли объект заданным свойством или был найден среди прототипов.

```
for (myvar in obj) {
    if (obj.hasOwnProperty(myvar)) {
        ...
    }
}
```



Инструкция `do` похожа на `while` за исключением того, что выражение проверяется после выполнения блока, а не до него. Это означает, что блок всегда будет выполнен хотя бы один раз.



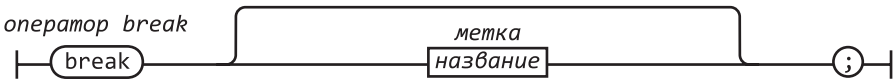
Инструкция `try` выполняет блок и перехватывает все вбрасываемые блоком исключения. В предложении `catch` определяется новая *переменная*, которая получит объект исключения.

Инструкция `throw` запускает исключение. Если инструкция `throw` содержит блок `try`, то управление передается в предложение `catch`. В противном случае выполнение функции прекращается, и управление передается в предложение `catch` инструкции `try` вызывавшей ее функции.

Здесь выражение — это, как правило, литерал объекта, содержащий свойства `name` и `message`. Перехватчик исключений может использовать эту информацию для управления дальнейшими действиями.

Инструкция `return` досрочно возвращает результат вызова функции. Также в ней можно указать возвращаемое значение. Если возвращаемое выражение не указано, то оно будет `undefined`.

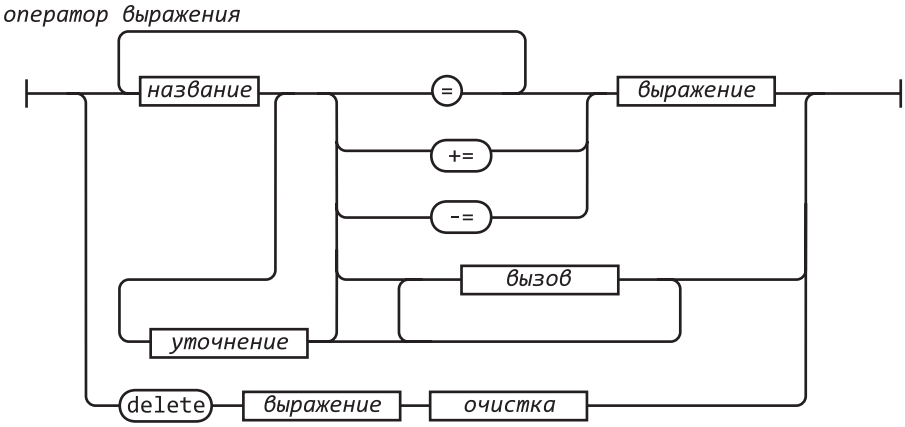
В JavaScript конец строки не может находиться между инструкцией `return` и выражением.



Инструкция `break` заставляет выйти из цикла или из инструкции `switch`. Опционально она может содержать метку, которая приведет к выходу из помеченной этой меткой инструкции.

В JavaScript конец строки не может находиться между инструкцией `break` и меткой.

Инструкция выражения позволяет присваивать значения одной или нескольких переменных или членов объекта, вызывать методы, удалять свойства объекта. Для присваивания используется оператор `=` . Не стоит путать его с оператором равенства `===`. Оператор `+=` применяется для сложения и конкатенации.



Выражения

Простейшее выражение может представлять собой литеральное значение (например, строку или число), переменную, встроенное значение (`true`, `false`, `null`, `undefined`, `NaN` или `Infinity`), вызывающее выражение, начинающееся с `new`, уточняющее выражение, начинающееся с `delete`, выражение, заключенное в круглые скобки, выражение, которому предшествует префиксный оператор, или выражение, за которым следует:

- инфиксный оператор и другое выражение;
- тернарный оператор `?`, за которым следует другое выражение, а затем после `:` еще одно;
- вызов;
- уточнение.

Тернарный оператор `?` принимает три операнда. Если первый операнд истинен, оператор принимает значение второго операнда. Если же первый операнд ложен, он принимает значение третьего операнда.

Операторы, находящиеся в верхней части табл. 2.1, имеют более высокий приоритет. Операторы в нижней части таблицы имеют низкий приоритет. Для изменения нормального приоритета могут быть использованы скобки:

```
2 + 3 * 5 === 17
(2 + 3) * 5 === 25
```

выражение

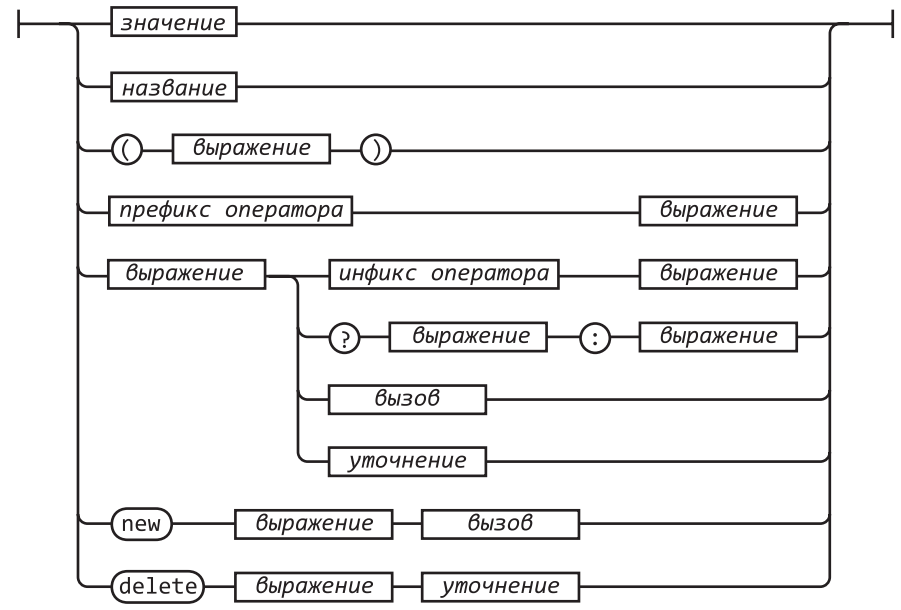
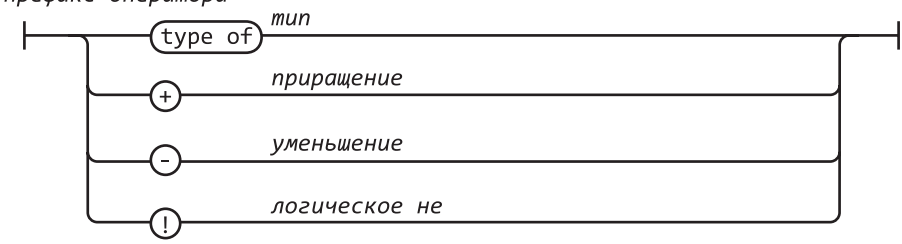


Таблица 2.1. Приоритеты операторов

Операторы	**Описание**
`. [] ()`	Очистка и вызов
`delete new typeof + - !`	Унарные операторы
`* / %`	Умножение, деление, получение остатка
`+ -`	Сложение (конкатенация), вычитание
`>= <= ><`	Неравенство
`=== !==`	Равенство
`&&`	Логическое И
`||`	Логическое ИЛИ
`?:`	Тернарный оператор

префикс оператора



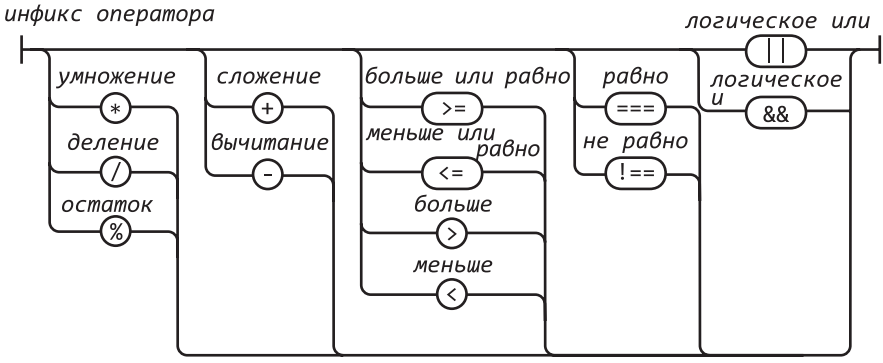
В операторе `typeof` могут использоваться значения `'number'`, `'string'`, `'boolean'`, `'undefined'`, `'function'` и `'object'`. Если операнд — массив или `null`, то результатом будет `'object'`, что неверно. Подробнее использование оператора `typeof` описано в главе 6 и приложении А.

Если операнд оператора `!` принимает значение `true`, то возвращается `false`. В противном случае он возвращает `true`.

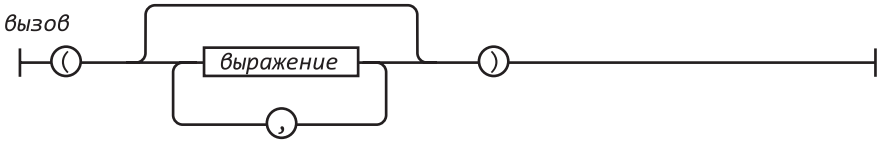
Оператор `+` выполняет сложение или конкатенацию. Если вы собираетесь использовать его для сложения, убедитесь, что оба операнда являются числами.

Оператор `/` может вернуть нецелый результат, даже если оба операнда являются целыми.

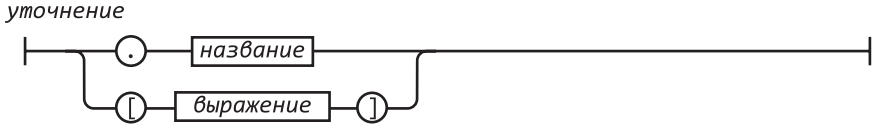
Оператор `&&` возвращает значение первого операнда, если первый операнд ложен. В противном случае он возвращает значение второго операнда.



Оператор `||` возвращает значение первого операнда, если первый операнд истинен. В противном случае он возвращает значение второго операнда.



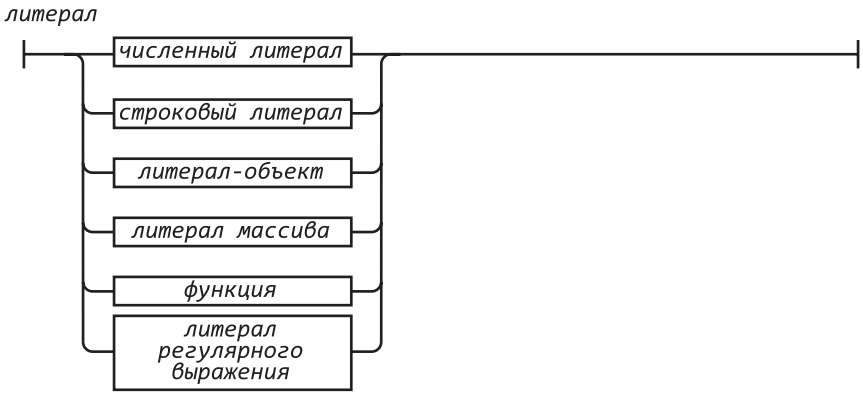
Вызовом функции является имя функции, за которым следует пара круглых скобок. В скобках могут содержаться аргументы, которые передаются в функцию. Подробнее о функциях рассказывается в главе 4.

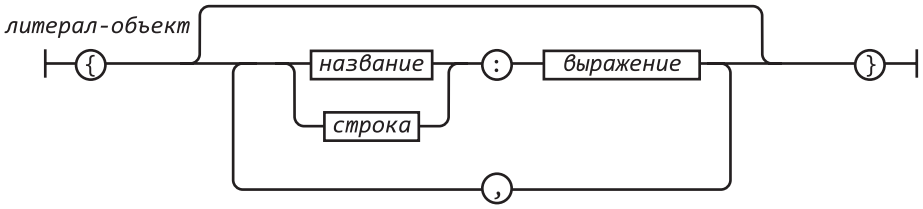


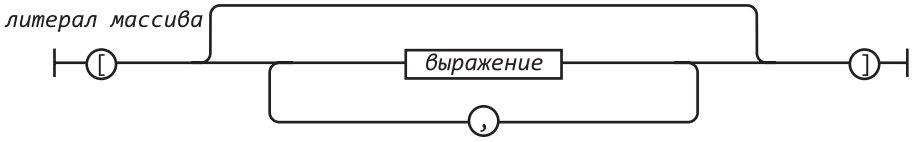
Уточнение позволяет задать свойство или элемент объекта или массива. В следующей главе об этом рассказывается более подробно.

Литералы

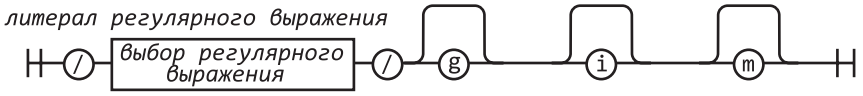
Литералы объектов представляют собой удобную нотацию для задания новых объектов. Названия свойств могут указываться как имена или как строки. Имена трактуются как литеральные имена, а не как имена переменных, поэтому имена свойств объекта должны быть известны на этапе компиляции. Значениями свойств являются выражения. Подробнее литералы-объекты описаны в следующей главе.







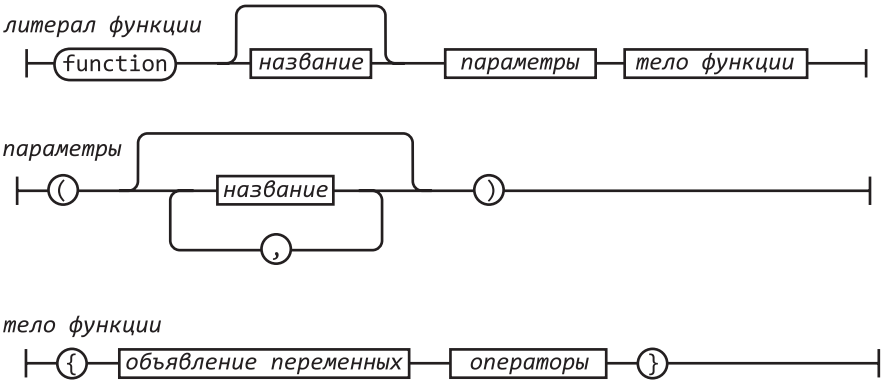
Для определения новых массивов удобно использовать литерал массива, который более подробно описан в главе 6.



О регулярных выражениях подробно рассказывается в главе 7.

Функции

Литерал функции определяет ее значение. Он может иметь имя (необязательно), которое дает возможность использовать рекурсивный вызов. Он может содержать список параметров, которые будут выступать в качестве аргументов функции. В теле функции находятся объявление переменных и инструкции. Подробнее о функциях рассказывается в главе 4.



3 Объекты

Любовь слепа на пошлые предметы.

Уильям Шекспир. Два веронца

Числа, строки, логические переменные (`true` и `false`), а также значения `null` и `undefined` в JavaScript относятся к простым типам данных. Все остальное является *объектами*. Числа, строки и логические переменные похожи на объекты тем, что имеют методы, но в отличие от объектов они неизменны. Объекты в JavaScript имеют изменяемые ключевые коллекции. В JavaScript объектами являются массивы, функции, регулярные выражения, и, конечно, объекты также являются объектами.

Объект представляет собой контейнер свойств, где каждое свойство имеет имя и значение. Именем свойства может быть любая строка, в том числе пустая. Значением свойства может быть любое значение, исключая `undefined`.

Объекты в JavaScript не привязаны к классам, поэтому не существует ограничений на именование новых свойств или на присваивание им значений. Объекты довольно удобно использовать для сбора и организации данных. Объекты могут содержать другие объекты, поэтому с их помощью можно легко представить структуру дерева или графа.

В JavaScript предусмотрена возможность прототипизирования сборки, позволяющая одному объекту наследовать свойства другого. При правильном использовании это может сократить время инициализации объекта и потребление памяти.

Литералы объектов

Литералы объектов представляют собой очень удобную нотацию для создания новых объектных значений. Литерал объекта задается парой фигурных скобок,

внутрь которых можно поместить пару имя/значение или оставить их пустыми. Литерал объекта может указываться везде, где может находиться выражение:

```
var empty_object = {};

var stooge = {
    "first-name": "Jerome",
    "last-name": "Howard"
};
```

Именем свойства может быть любая строка, включая пустую. Кавычки вокруг имени свойства в литерале объекта указывать не обязательно, если это имя допустимо в JavaScript и не является зарезервированным словом. Таким образом, кавычки необходимы в имени `"first-name"`, но не обязательны для `first_name`. Для разделения пар имя/значение используются запятые.

Значение свойства может быть получено из любого выражения, в том числе из другого литерала объекта. Объекты могут быть вложенными:

```
var flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42", city: "Los Angeles"
    }
};
```

Получение значений

Получить значения свойств объекта можно, заключив строку в квадратные скобки `[]`. Если строка является строковым литералом, допустима в JavaScript и не является зарезервированным словом, то вместо квадратных скобок можно использовать точку (`.`). Поскольку нотация с точкой компактнее и лучше читается, она считается более предпочтительной:

```
stooge["first-name"]     // "Jerome"
flight.departure.IATA    // "SYD"
```

При попытке получить несуществующий член объекта возвращается значение `undefined`:

```
stooge["middle-name"]    // undefined
flight.status            // undefined
stooge["FIRST-NAME"]     // undefined
```

Для задания значений, предлагаемых по умолчанию, может быть использован оператор ||:

```
var middle = stooge["middle-name"] || "(none)";
var status = flight.status || "unknown";
```

При попытке получения значения undefined вбрасывается исключение TypeError. Этого можно избежать, воспользовавшись оператором &&:

```
flight.equipment                              // undefined
flight.equipment.model                        // вброс "TypeError"
flight.equipment && flight.equipment.model    // undefined
```

Обновление

Значение объекта может быть обновлено путем присваивания. Если свойство с таким именем уже существует в объекте, его значение заменяется:

```
stooge['first-name'] = 'Jerome';
```

Если объект еще не имеет свойства с таким именем, оно добавляется к объекту:

```
stooge['middle-name'] = 'Lester';
stooge.nickname = 'Curly';
flight.equipment = {
    model: 'Boeing 777'
};
flight.status = 'overdue';
```

Ссылки

Объекты передаются по ссылке. Они никогда не копируются:

```
var x = stooge;
x.nickname = 'Curly';
var nick = stooge.nickname;
    // nick имеет значение 'Curly', так как x и stooge
    // являются ссылками на один и тот же объект
var a = {}, b = {}, c = {};
    // каждый из объектов a, b и c – это ссылка на
    // новый пустой объект
a = b = c = {};
    // все объекты a, b и c являются ссылками на
    // один и тот же объект
```

Прототипы

Каждый объект связан с объектом-прототипом, от которого он может наследовать свойства. Все объекты, создаваемые указанием литералов объектов, связаны с объектом Object.prototype, который обычно поставляется вместе с JavaScript.

При создании нового объекта можно выбрать объект, который будет его прототипом. Механизм, обеспечивающий подобную возможность в JavaScript, реализован довольно запутанно и неаккуратно, но может быть значительно упрощен. Для этого к функции `Object` нужно добавить метод `create`. Метод `create` создает новый объект, использующий старый объект в качестве прототипа. Подробнее функции описаны в следующей главе.

```
if (typeof Object.create !== 'function') {
    Object.create = function (o) {
        var F = function () {};
        F.prototype = o;
        return new F();
    };
}
var another_stooge = Object.create(stooge);
```

Связывание объекта с прототипом никак не влияет на обновление. В случае если в объект вносятся изменения, то они не касаются прототипа объекта:

```
another_stooge['first-name'] = 'Harry';
another_stooge['middle-name'] = 'Moses';
another_stooge.nickname = 'Moe';
```

Ссылка на прототип нужна только для получения значения. Если мы начнем получать значение свойства, которое отсутствует у объекта, то JavaScript попытается получить значение этого свойства у объекта-прототипа. А если его не будет и у этого объекта, то он, в свою очередь, обратится к своему прототипу и т. д., пока, наконец, дело не дойдет до объекта `Object.prototype`. Если требуемого свойства не существует в цепочке прототипов, то результатом будет значение `undefined`. Этот процесс называется делегированием.

Прототипы имеют динамические отношения. Если добавить прототипу новое свойство, то оно немедленно станет видимым для всех объектов, производных от этого прототипа:

```
stooge.profession = 'actor';
another_stooge.profession    // 'actor'
```

Подробнее цепочки прототипов описаны в главе 6.

Отражения

Можно легко проверить, существуют ли свойства объекта, которые он пытается получить, и изучить полученные значения. Оператор `typeof` может быть очень полезным при определении типа свойства:

```
typeof flight.number      // 'number'
typeof flight.status      // 'string'
typeof flight.arrival     // 'object'
typeof flight.manifest    // 'undefined'
```

Необходимо соблюдать некоторую осторожность, поскольку любое свойство цепочке прототипов может дать значение:

```
typeof flight.toString    // 'function'
typeof flight.constructor // 'function'
```

Существует два подхода к решению проблемы несуществующих свойств. Во-первых, создание собственной программы с функциями получения и исключения нежелательных значений свойств. Когда используется отражение, важны данные, поэтому нельзя забывать, что некоторые свойства могут быть функциями.

Другой подход заключается в использовании метода `hasOwnProperty`, который возвращает значение `true`, если объект обладает заданным свойством. Метод `hasOwnProperty` не обращается к цепочке прототипов:

```
flight.hasOwnProperty('number')            // true
flight.hasOwnProperty('constructor')       // false
```

Перечисление

В цикле `for in` можно задать имена всех свойств объекта. Перечисление будет включать в себя все свойства, в том числе функции и свойства прототипов, которые не всегда важны, но такие значения можно отфильтровать. Наиболее распространенные фильтры — метод `hasOwnProperty` и функция `TypeOf`:

```
var name;
for (name in another_stooge) {
    if (typeof another_stooge[name] !== 'function') {
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

Будьте готовы к тому, что свойства могут появляться в любом порядке. Если вы хотите, чтобы свойства появлялись в определенном порядке, то лучше избегать объявления объекта целиком и вместо этого создать массив, содержащий имена свойств в нужном порядке:

```
var i;
var properties = [
    'first-name',
    'middle-name',
    'last-name',
    'profession'
];
for (i = 0; i < properties.length; i += 1) {
    document.writeln(properties[i] + ': ' +
        another_stooge[properties[i]]);
}
```

Используя цикл `for in`, можно получить необходимые свойства, не беспокоясь о том, что они будут извлечены из цепочки прототипов, причем получить их можно в нужном порядке.

Удаление

Оператор `delete` может быть использован для удаления свойств объекта. Он удаляет свойство объекта, если оно есть. Удаление не касается свойств объектов, связанных по цепочке прототипов.

Удаление свойства объекта дает возможность использовать значение одноименного свойства объекта-прототипа:

```
another_stooge.nickname    // 'Moe'

// После удаления свойства nickname объекта another_stooge
// станет доступно свойство nickname объекта-прототипа

delete another_stooge.nickname;

another_stooge.nickname        // 'Curly'
```

Ослабление глобальности

JavaScript позволяет легко определять глобальные переменные, содержащие все переменные вашего приложения. К сожалению, глобальные переменные делают программы менее устойчивыми, поэтому их следует избегать.

Одним из способов минимизировать использование глобальных переменных является создание в приложении единственной глобальной переменной:

```
var MYAPP = {};
```

Эта переменная становится контейнером вашего приложения:

```
MYAPP.stooge = {
    "first-name": "Joe",
    "last-name": "Howard"
};
MYAPP.flight = {
    airline: "Oceanic",
    number: 815,
    departure: {
        IATA: "SYD",
        time: "2004-09-22 14:55",
        city: "Sydney"
    },
```

```
    arrival: {
        IATA: "LAX",
        time: "2004-09-23 10:42",
        city: "Los Angeles"
    }
};
```

За счет использования одной глобальной переменной можно значительно снизить вероятность неудачного взаимодействия с другими приложениями, виджетами или библиотеками. Программа станет более понятной, так как очевидно, что `MYAPP.stooge` ссылается на высший уровень структуры. В следующей главе рассказывается о способах использования замыканий для скрытия информации, что является еще одним эффективным методом «ослабления глобальности».

4 Функции

Но каждый грех еще до совершенья
Уж осужден. Обязанность свою
Я обратил бы в нуль...

Уильям Шекспир. Мера за меру

Лучше всего в JavaScript реализованы функции. В них почти все правильно. Почти, но не все, как, впрочем, и следовало бы ожидать от JavaScript.

Каждая функция состоит из группы инструкций. Функции — фундаментальный модульный элемент JavaScript. Они необходимы для многократного использования одних и тех же фрагментов кода, скрытия информации и построения программ. Функции определяют поведение объектов. Основная задача программирования — это перевод поставленных требований в функции и структуры данных.

Объекты функций

Функции в JavaScript являются объектами. Объекты представляют собой коллекции пар имя/значение, имеющие скрытую ссылку на прототип объекта. Объекты, созданные посредством литералов объектов, связаны с объектом `Object.prototype`. Объекты функций связаны с объектом `Function.prototype` (который, в свою очередь, связан с `Object.prototype`). Кроме того, каждая функция создается с двумя дополнительными скрытыми свойствами: контекстом функции и кодом, реализующим поведение функции.

При создании каждый объект функции получает свойство `prototype`. Значением этого свойства является объект, имеющий свойство `constructor`, значение которого — функция. Это совсем не похоже на скрытую связь с объектом `Function.prototype`. Суть данной запутанной схемы раскрывается в следующей главе.

Поскольку функции являются объектами, они могут использоваться как любые другие переменные. Функции могут храниться в переменных, объектах и массивах.

Их можно передавать в качестве аргументов, а также возвращать в качестве результата вызова других функций. Кроме того, поскольку функции являются объектами, они могут иметь методы.

Отличительной особенностью функций является возможность их вызова.

Литералы функций

Объекты функций создаются с помощью литералов функций:

```
// Создаем переменную add для хранения
// функции сложения двух чисел.
var add = function (a, b) {
    return a + b;
};
```

Литерал функции состоит из четырех частей. Первая часть — это зарезервированное слово `function`.

Необязательная вторая часть — имя функции. Функция может задействовать свое имя для рекурсивного вызова. Кроме того, имя может использоваться отладчиками и средствами разработки для идентификации функции. Если у функции нет имени, как в предыдущем примере, она называется *анонимной*.

Третья часть — это набор параметров функции, заключенный в скобки. В скобках могут находиться несколько параметров, разделенных запятыми, или же параметров может не быть вообще. Имена параметров определяются в функции как переменные. В отличие от обычных переменных, они инициализируется не значением `undefined`, а аргументами, получаемыми функцией при ее вызове.

Четвертая часть представляет собой набор инструкций, заключенных в фигурные скобки. Эти инструкции составляют тело функции и выполняются при ее вызове.

Литерал функции может располагаться в любом месте кода, где допустимы выражения. Функции могут определяться внутри других функций. Естественно, что внутренняя функция имеет доступ к своим параметрам и переменным. Кроме того, она будет иметь доступ к параметрам и переменным вызвавшей ее функции. Объект функции, созданный с помощью литерала функции, содержит ссылку на внешний контекст. Это называется *замыканием*. Замыкание — это источник огромной силы.

Вызовы

Вызов функции приостанавливает выполнение текущей функции и передает управление и параметры новой функции. В дополнение к объявленным параме-

трам каждая функция получает два дополнительных параметра: `this` и `arguments`. Эти параметры очень важны в объектно-ориентированном программировании, и их значение определяется *шаблоном вызова*. В JavaScript поддерживаются четыре шаблона вызова: шаблон вызова метода, шаблон вызова функции, шаблон вызова конструктора и шаблон вызова метода `apply`. Шаблоны отличаются способом инициализации параметра `this`.

Оператор вызова представляет собой пару круглых скобок с любым выражением, возвращающим значение функции. Скобки могут содержать несколько выражений, разделенных запятыми, или не содержать их вообще. Результатом каждого выражения является значение одного аргумента. Каждое из значений аргументов присваивается имени параметра функции. Даже в том случае, если количество аргументов не соответствует количеству параметров функции, ошибки времени выполнения не возникает. Если значений аргументов слишком много, лишние значения аргументов игнорируются. Если слишком мало, недостающие значения заменяются значением `undefined`. Проверки типа значения аргументов не происходит: любому параметру может быть передано значение любого типа.

Шаблон вызова метода

Когда функция хранится как свойство объекта, она называется методом. Когда вызывается метод, параметр `this` связывается с этим объектом. Если выражение вызова содержит уточнение, то есть точку или выражение [*индекс*], то оно вызывается как метод:

```
// Создаем объект MyObject. Он имеет значение и метод приращения.
// Метод приращения принимает дополнительные параметры.
// Если аргумент не является числом, то
// по умолчанию имеет значение 1.

var myObject = { value: 0,
    increment: function (inc) {
        this.value += typeof inc === 'number' ? inc : 1;
    }
};

myObject.increment( );
document.writeln(myObject.value);    // 1

myObject.increment(2);
document.writeln(myObject.value);    // 3
```

Методы можно использовать для доступа к объекту, с их помощью можно изменять объект или получать его значения. Связывание объекта с `this` происходит во время вызова. Такое позднее связывание дает больше возможностей для многократного использования функции, имеющей параметр `this`. Методы, которые получают контекст объекта от `this`, называются *открытыми*.

Шаблон вызова функции

Если функция не является свойством объекта, она вызывается как функция:

```
var sum = add(3, 4);    // sum имеет значение 7
```

Когда функция вызывается таким способом, `this` связывается с глобальным объектом. Это ошибка в конструкции языка. Если бы JavaScript был разработан правильно, то при вызове внутренней функции параметр `this` был бы по-прежнему привязан к переменной `this` внешней функции. Следствием этой ошибки является то, что метод не может использовать внутреннюю функцию, которая помогла бы ему решить поставленную задачу, потому что внутренняя функция не разделяет доступ метода к объекту, так как `this` связывается с неправильной переменной. К счастью, есть простое решение этой проблемы. Если метод определяет переменную и присваивает ей значение `this`, внутренняя функция получает доступ к `this` через эту переменную. Общепринятое имя этой переменной — `that`:

```
// Дополняем MyObject методом double.

myObject.double = function () {
    var that = this;    // Решение.

    var helper = function () {
        that.value = add(that.value, that.value);
    };

    helper( );          // Вызов функции helper.
};

// Вызов метода double.

myObject.double( );
document.writeln(myObject.value);    //6
```

Шаблон вызова конструктора

JavaScript — язык наследования *прототипов*. Это означает, что объекты могут наследовать свойства непосредственно от других объектов. Это язык без классов.

Вот такое радикальное отступление от современной моды. Большинство языков сегодня поддерживают *классы*. Наследование прототипов — довольно мощный, но не общепризнанный механизм. Похоже, язык JavaScript сам не очень уверен в своей прототипизированной природе, поэтому его синтаксис создания объектов напоминает классические языки. Не многие из классических программистов считают приемлемым наследование прототипов, к тому же схожесть с классическим синтаксисом скрывает истинную природу прототипизированного языка, и это хуже всего.

Если функция вызывается с префиксом `new`, то новый объект создается со скрытой связью со значением члена `prototype` функции, и параметр `this` связывается с этим новым объектом.

Кроме того, префикс `new` меняет поведение инструкции `return`. Подробнее об этом рассказывается далее.

```
// Создаем функцию конструктора с именем Quo.
// Она создает объект со свойством status.

var Quo = function (string) {
    this.status = string;
};

// Передаем все экземпляры Quo открытому
// методу get_status.

Quo.prototype.get_status = function () {
    return this.status;
};

// Создаем экземпляр Quo.

var myQuo = new Quo("confused");

document.writeln(myQuo.get_status( ));
```

Функции, предназначенные для использования с префиксом `new`, называются *конструкторами*. По соглашению они хранятся в переменных, имена которых начинаются с прописной буквы. Если вызвать конструктор без префикса `new`, могут произойти очень плохие вещи, причем без всяких предупреждений на этапе компиляции или выполнения, так что использовать прописные буквы действительно важно.

Применять подобный подход не рекомендуется. Более удачные альтернативы описаны в следующей главе.

Шаблон вызова метода apply

Поскольку JavaScript — это функциональный объектно-ориентированный язык, функции могут иметь методы.

Метод `apply` позволяет создать массив аргументов, используемых для вызова функции. Также он позволяет выбирать значение `this`. Метод `apply` принимает два параметра. Первый — переменная, которая должна быть связана с `this`. Второй — массив параметров.

```
// Создает массив из двух чисел и складывает их.
var array = [3, 4];
var sum = add.apply(null, array);    // sum имеет значение 7
```

```
// Создает объект, имеющий свойство status.

var statusObject = {
    status: 'A-OK'
};

// statusObject не наследуется от Quo.prototype,
// но может вызывать метод get_status для
// statusObject, хотя statusObject не имеет
// метода get_status.

var status = Quo.prototype.get_status.apply(statusObject);
    // status имеет значение 'A-OK'
```

Аргументы

Дополнительный параметр, доступный функции во время вызова, представляет собой массив `arguments`. Функция имеет доступ ко всем аргументам, объявленным во время ее вызова, в том числе к лишним аргументам, которым не были назначены параметры. Это позволяет писать функции, принимающие неопределенное количество параметров:

```
// Создаем функцию, добавляющую данные.

// Обратите внимание, что определение переменной sum внутри
// функции не влияет на значение sum,
// определеное вне функции. Функция видит
// только внутреннюю переменную.

var sum = function () {
    var i, sum = 0;
    for (i = 0; i < arguments.length; i += 1) {
        sum += arguments[i];
    }
    return sum;
};

document.writeln(sum(4, 8, 15, 16, 23, 42));     // 108
```

Это не особенно полезный шаблон. О том, как добавить аналогичный метод для массива, рассказывается в главе 6.

Из-за ошибки проектирования `arguments` — это, на самом деле, не массив, а объект подобный массиву. `arguments` имеет свойство `length`, но ему не хватает методов массива. К чему может привести подобная ошибка проектирования, описано в конце этой главы.

Возвращение управления из функции

При вызове функции она выполняется, начиная с первой инструкции, и заканчивается после фигурной скобки, закрывающей тело функции. В этот момент функция возвращает управление той части программы, которая ее вызвала.

Инструкция `return` служит для преждевременного завершения функции. При этом функция немедленно возвращает управление без выполнения оставшихся инструкций.

Функция всегда возвращает значение. Если возвращаемое значение не задано, то она возвращает значение `undefined`.

Если функция вызывается с префиксом `new`, а возвращаемым значением `return` не является объект, то вместо него возвращается `this` (новый объект).

Исключения

В JavaScript предусмотрен механизм обработки исключений. Исключениями являются необычные (но не совсем неожиданные) ошибки, которые мешают нормальному выполнению программы. В случае их обнаружения программа должна сгенерировать исключение:

```
var add = function (a, b) {
    if (typeof a !== 'number' || typeof b !== 'number') {
        throw {
            name: 'TypeError',
            message: 'add needs numbers'
        };
    }
    return a + b;
}
```

Инструкция `throw` прерывает выполнение функции. Ей передается объект исключения, содержащий свойство `name`, которое определяет тип исключения, и свойство `message`, необходимое для описания исключения. Также можно добавить и другие свойства.

Далее объект исключения попадает в соответствующую ветвь `catch` инструкции `try`:

```
// Создаем функцию try_it, некорректно вызывающую
// новую функцию add.
var try_it = function () {
    try {
        add("seven");
    } catch (e) {
        document.writeln(e.name + ': ' + e.message);
    }
}
try_it();
```

Если исключение генерируется внутри блока `try`, управление передается в предложение `catch`.

В каждой инструкции `try` одна из ветвей `catch` предназначается для перехвата всех исключений. Если вариант обработки зависит от типа исключения, то обработчик будет проверять свойство `name`, чтобы определить тип исключения.

Расширенные типы

JavaScript позволяет расширить основные типы языка. В главе 3 говорилось, что добавление метода в объект `Object.prototype` делает этот метод доступным всем объектам. Этот подход работает также для функций, массивов, строк, чисел, регулярных выражений и логических переменных.

Например, расширив объект `Function.prototype`, можно создать метод, доступный для всех функций:

```
Function.prototype.method = function (name, func) {
    this.prototype[name] = func;
    return this;
};
```

Расширяя объект `Function.prototype` методом `method`, не нужно задавать имя свойства `prototype`. Теперь можно скрыть это небольшое безобразие.

JavaScript не имеет отдельного целого типа, но иногда требуется получить только целую часть числа. В JavaScript соответствующий метод реализован не очень красиво. Однако это можно исправить, добавив в `Number.prototype` метод `integer`. Он использует либо `Math.ceil`, либо `Math.floor`, в зависимости от знака числа:

```
Number.method('integer', function ( ) {
    return Math[this < 0 ? 'ceil' : 'floor'](this);
});

document.writeln((-10 / 3).integer());    // -3
```

В JavaScript не хватает метода, который удалял бы пробелы в конце строки. Это можно легко исправить следующим образом:

```
String.method('trim', function () {
    return this.replace(/A\s+|\s+$/g, '');
});

document.writeln('"' + " neat ".trim() + '"');
```

Метод `trim` использует регулярные выражения. Подробнее о регулярных выражениях рассказывается в главе 7.

Расширяя основные типы, можно значительно усилить выразительность языка. Из-за динамичного характера прототипизированного наследования в JavaScript

все значения сразу наделены новыми методами, даже значения, которые были созданы до создания методов.

Прототипы базовых типов — это открытые структуры, поэтому необходимо соблюдать осторожность при смешивании библиотек. Один из способов защиты — это добавление следующего метода, если его, конечно, не хватает:

```
// Добавляем условный метод.
Function.prototype.method = function (name, func) {
    if (!this.prototype[name]) {
        this.prototype[name] = func;
        return this;
    }
};
```

Еще одной проблемой является то, что инструкция цикла `for in` плохо взаимодействует с прототипами. Как было сказано в главе 3, для сглаживания этого недостатка можно использовать метод `hasOwnProperty`, отсекающий наследуемые свойства, или искать свойства конкретного типа.

Рекурсия

Рекурсивной называется функция, которая прямо или косвенно вызывает саму себя. Рекурсия — мощное средство программирования, где проблема делится на множество подобных подзадач, каждая из которых решается довольно просто. Как правило, рекурсивная функция вызывает саму себя для решения своих подзадач.

Ханойская башня — довольно известная головоломка. В ней имеются три штыря и набор дисков различного диаметра с отверстиями в центре. Диски собраны в пирамиду на одном из штырей, причем диски меньшего размера лежат на бóльших. Задача состоит в том, чтобы полностью переместить пирамиду на другой штырь, причем перемещать диски нужно по одному, а класть больший диск на меньший запрещено. У этой головоломки есть тривиальное рекурсивное решение:

```
var hanoi = function hanoi(disc, src, aux, dst) {
    if (disc > 0) {
        hanoi(disc - 1, src, dst, aux);
        document.writeln('Move disc ' + disc +
            ' from ' + src + ' to ' + dst);
        hanoi(disc - 1, aux, src, dst);
    }
};

hanoi(3, 'Src', 'Aux', 'Dst');
```

Вот решение для трех дисков:

```
Перемещаем диск 1 с Src на Dst
Перемещаем диск 2 с Src на Aux
Перемещаем диск 1 с Dst на Aux
Перемещаем диск 3 с Src на Dst
Перемещаем диск 1 с Aux на Src
Перемещаем диск 2 с Aux на Dst
Перемещаем диск 1 с Src на Dst
```

Функция `hanoi` перемещает стопки дисков с одного штыря на другой, используя третий, если это необходимо, что разбивает задачу на три подзадачи. Во-первых, она освобождает нижний диск, перемещая вышестоящие на другие позиции. Затем он становится нижним диском новой пирамиды. Наконец, на него последовательно перемещаются остальные диски. Такое перемещение вызывается рекурсивно для каждой подзадачи.

В функцию `hanoi` передается количество дисков, которые необходимо переместить, и три возможные позиции. Когда она вызывает себя, она должна работать с диском, который находится над текущим. В конце концов, она будет вызвана с несуществующего диска. В таком случае она ничего не делает. Это говорит о том, что рекурсия прекращается.

Рекурсивные функции могут быть очень эффективны при манипулировании древовидными структурами, такими как объектная модель документа (DOM) браузера. Каждый рекурсивный вызов обрабатывает меньшую часть дерева:

```
// Определяем функцию walk_the_DOM, которая обходит каждый
// узел дерева в HTML-документе по порядку, начиная
// с некоторого заданного узла. Она вызывает функцию, в которую,
// в свою очередь, передает следующий узел. walk_the_DOM вызывает
// саму себя для обработки дочерних узлов.

var walk_the_DOM = function walk(node, func) {
    func(node);
    node = node.firstChild;
    while (node) {
        walk(node, func);
        node = node.nextSibling;
    }
};

// Определяем функцию getElementsByAttribute. Она
// принимает строку имени атрибута и, необязательно,
// соответствующее значение. Она вызывает функию walk_the_DOM,
// передавая ей функцию, которая ищет имя атрибута в узле.
// Найденные узлы собираются в массиве результатов.

var getElementsByAttribute = function (att, value) {
    var results = [];
```

продолжение ⇩

```
    walk_the_DOM(document.body, function (node) {
        var actual = node.nodeType === 1 && node.getAttribute(att);
        if (typeof actual === 'string' &&
                (actual === value || typeof value !== 'string')) {
            results.push(node);
        }
    });

    return results;
};
```

Некоторые языки предлагают *оптимизацию хвостовой рекурсии*. Это означает, что если функция завершается рекурсивным вызовом самой себя, то вызов заменяется циклом, что может значительно ускорить процесс. К сожалению, JavaScript в настоящее время не обеспечивают оптимизацию хвостовой рекурсии. Функции, уходящие в глубокую рекурсию, в ходе обработки стека возврата могут завершиться неудачей:

```
// Делаем функцию factorial с хвостовой
// рекурсией. Это будет хвостовая рекурсия, потому что
// функция возвращает в качестве результат вызов самой себя.

// JavaScript в настоящее время не позволяет оптимизировать эту форму.

var factorial = function factorial(i, a) {
    a = a || 1;
    if (i < 2) {
        return a;
    }
    return factorial(i - 1, a * i);
};

document.writeln(factorial(4));    // 24
```

Область видимости

Область видимости в языке программирования определяет видимость и время жизни переменных и параметров. Это важное для программистов понятие, которое помогает избежать конфликтов имен и обеспечивает автоматическое управление памятью:

```
var foo = function () {
    var a = 3, b = 5;

    var bar = function () {
        var b = 7, c = 11;
// В этот момент a равно 3, b равно 7, c равно 11

        a += b + c;
```

```
// В этот момент a равно 21, b равно 7, c равно 11

    };
// В этот момент a равно 3, b равно 5, c не определено

    bar( );
// В этот момент a равно 21, b равно 5

};
```

Большинство языков, синтаксис которых напоминает синтаксис языка C, поддерживает понятие области видимости блока. Все переменные, определенные в некотором блоке (в перечне инструкций, заключенном в фигурные скобки), не видны извне блока. Переменные, определенные в блоке, могут быть освобождены, когда выполнение блока завершится. Это — хорошая вещь.

К сожалению, в JavaScript понятие области видимости блока не поддерживается, хотя, глядя на синтаксис блока инструкций, кажется, что должно поддерживаться. Подобная путаница может стать источником ошибок.

В то же время в JavaScript поддерживается понятие области видимости функции. Это означает, что параметры и переменные, определенные в функции, не видны за пределами функции, а переменная, определенная в любой точке функции, видна повсюду в пределах функции.

Во многих современных языках рекомендуется объявлять переменные как можно позже, при их первом использовании. Но для языка JavaScript это оказывается плохим советом, поскольку в нем отсутствует понятие области видимости блока. Вместо этого все переменные, используемые в функции, лучше всего объявлять в начале тела функции.

Замыкания

Положительный момент, связанный с областью видимости функций, состоит в том, что внутренние функции получают доступ к параметрам и переменным функций, в которых они определены (за исключением параметров `this` и `arguments`), — это очень полезная вещь.

Функция `getElementsByAttribute` работает, поскольку в ней объявлена переменная `result`, и внутренняя функция `walk_the_DOM` также имеет доступ к переменной `result`.

Гораздо интереснее случай, когда внутренняя функция имеет более долгое время жизни, чем внешняя.

Ранее был создан объект `MyObject`, имеющий методы `value` и `increment`. Предположим, что необходимо защитить значения свойств объекта от несанкционированных изменений.

Вместо того чтобы инициализировать MyObject с помощью литерала объекта, нужно инициализировать MyObject, вызвав функцию, возвращающую литерал объекта. В этой функции определяется переменная value, для которой всегда доступны методы increment и getValue, но область видимости функции скрывает ее от остальной части программы:

```
var myObject = (function () {
    var value = 0;
    return {
        increment: function (inc) {
            value += typeof inc === 'number' ? inc : 1;
        },
        getValue: function () {
            return value;
        }
    };
}());
```

В этом примере объект MyObject не присваивается функции. Ему присваивается результат вызова этой функции. Обратите внимание на круглые скобки ( ) в последней строке. Функция возвращает объект, содержащий два метода, и эти методы продолжают пользоваться привилегиями доступа к переменной value.

Ранее в этой главе был описан конструктор Quo, который создает объект, обладающий свойством status и методом get_status. Но это, кажется, было не очень интересно. Почему метод получения свойства имеет доступ к его значению? Было бы полезнее, если бы свойство status было закрытым? Итак, давайте определим различные варианты функции quo, чтобы достигнуть поставленной цели:

```
// Создаем функцию с именем quo. Это делает объект
// с методом get_status и закрытым свойством status.
var quo = function (status) {
    return {
        get_status: function ( ) {
            return status;
        }
    };
};

// Создаем экземпляр quo.

var myQuo = quo("amazed");
document.writeln(myQuo.get_status());
```

Функция quo спроектирована так, что при ее использовании не требуется префикс new, поэтому ее имя не нужно писать с заглавной буквы. При вызове quo она возвращает новый объект, содержащий метод get_status. В myQuo хранится ссылка на этот объект. Метод get_status имеет привилегированный доступ к свойству status функции quo, хотя функция quo уже вернула управление. Метод get_status

имеет доступ не к копии параметра, а к самому параметру. Такое возможно, поскольку функции доступен контекст, в котором она была создана. Это и называется *замыканием*.

Рассмотрим более полезный пример:

```
// Определяем функцию, которая устанавливает желтый цвет
// для узла DOM, а затем заменяет его белым.

var fade = function (node) {
    var level = 1;
    var step = function () {
        var hex = level.toString(16);
        node.style.backgroundColor = '#FFFF' + hex + hex;
        if (level < 15) {
            level += 1;
            setTimeout(step, 100);
        }
    };
    setTimeout(step, 100);
};
fade(document.body);
```

Мы вызываем функцию `fade`, в которую передаем параметр `document.body` (узел, созданный HTML-тегом `<body>`). Функция `fade` устанавливает для `level` значение 1, которое определяется функцией `step`. Она вызывает метод `setTimeout`, передавая ему функцию `step` и время (100 миллисекунд). После этого она возвращает управление — `fade` завершается.

Внезапно, около 10 секунд спустя, вызывается функция `step`. Она задает базу из 16 символов из параметра `level` функции `fade`. Затем она изменяет цвет фона узла `fade`. Далее она ищет значение `level` из функции `fade`. Если цвет еще не белый, то `level` пошагово увеличивается и использует `setTimeout`, чтобы запланировать свой очередной запуск.

Вдруг функция `step` вызывается снова. Но на этот раз значение `level` функции `fade` равно 2. Функция `fade` возвращает результат, но ее переменные продолжают существовать до тех пор, пока они используются хотя бы одной функцией внутри `fade`.

Важно понимать, что внутренняя функция имеет доступ к реальным переменным внешней функции, а не к их копиям, благодаря чему можно избежать следующей проблемы:

```
// ПЛОХОЙ ПРИМЕР
// Делаем функцию, которая присваивает функции обработчика события
// массив узлов (неверный путь). При щелчке на узле в окне
// предупреждения должен отображаться порядковый номер узла.
// Но вместо этого там всегда отображается число узлов.
```

продолжение ⇩

```
var add_the_handlers = function (nodes) {
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = function (e) {
            alert(i);
        };
    }
};
// ОКОНЧАНИЕ ПЛОХОГО ПРИМЕРА
```

Функция `add_the_handlers` должна была присваивать каждому обработчику уникальный номер `i`. Ничего не получилось потому, что обработчик функции связывается с переменной `i`, а не со значением переменной `i` в момент создания функции.

```
// Лучший пример

// Делаем функцию, которая присваивает функции обработчика события
// массив узлов. При щелчке на узле в окне предупреждения
// отображается порядковый номер узла.

var add_the_handlers = function (nodes) {
    var helper = function (i) {
        return function (e) {
            alert(i);
        };
    };
    var i;
    for (i = 0; i < nodes.length; i += 1) {
        nodes[i].onclick = helper(i);
    }
};
```

Следует избегать создания функций внутри цикла, так как это может усложнить вычисления и привести к путанице, как произошло в плохом примере. Избежать путаницы поможет функция `helper`, созданная вне цикла, которая возвращает функцию, связанную с текущим значением `i`.

Обратные вызовы

Функции облегчают обработку дискретных событий. К примеру, представим следующую последовательность: пользователь отправляет запрос серверу и получает ответ. Было бы наивно полагать, что она описывается следующим образом:

```
request = prepare_the_request();
response = send_request_synchronously(request);
display(response);
```

Недостаток данного подхода в том, что синхронный запрос по сети оставляет клиента в ожидании ответа от сервера. Если сеть или сервер работает медленно, время отклика возрастает в разы.

В этом случае лучше использовать асинхронный запрос, обеспечивающий выполнение функции обратного вызова после получения ответа от сервера. Асинхронная функция немедленно возвращает значение, так что клиент не блокируется в ожидании ответа, как в предыдущем случае:

```
request = prepare_the_request();
send_request_asynchronously(request, function (response) {
    display(response);
});
```

В функцию `send_request_asynchronously` в качестве параметра передается функция, которая вызывается после получения ответа от сервера.

Модули

Для создания модулей используются функции и замыкания. Модуль представляет собой функцию или объект, который предоставляет интерфейс, но скрывает свое состояние и реализацию. При использовании функций для создания модулей почти полностью исключается применение глобальных переменных, тем самым сглаживается один из главных недостатков JavaScript.

Например, предположим, что необходимо изменить строку с помощью метода `deentityify`, задача которого состоит в поиске HTML-элементов в строке и замене их эквивалентными символами. Имеет смысл хранить названия элементов и их эквиваленты в объекте. Но где же хранить сам объект? Можно хранить его в глобальной переменной, но это — не лучшее решение. Можно определить объект в самой функции, но это чревато увеличением времени выполнения программы, так как объект будет обрабатываться при каждом вызове функции. Идеальным решением является хранение его в локальной переменной внутри замыкания, также, возможно, потребуется определить метод, позволяющий добавлять дополнительные объекты:

```
String.method('deentityify', function () {

// Таблица сущностей. Отображает имена сущностей на символы.

var entity = {
    quot: '"',
    lt:   '<',
    gt:   '>'
};
```

продолжение ⇩

```
// Возвращаем метод deentityify.
    return function () {

// Метод deentityify. Метод строки вызывается для замены найденной
// подстроки, начинающейся с символа '&' и заканчивающейся символом ';'.
// Если символы внутри есть в таблице сущностей, то эти элементы
// заменяются символами из таблицы. Метод используется
// в регулярных выражениях (см. главу 7).

        return this.replace(/&([A&;]+);/g,
            function (a, b) {
                var r = entity[b];
                return typeof r === 'string' ? r : a;
            }
        );
    };
}());
```

Обратите внимание на последнюю строку. В ней функция вызывается с помощью оператора `()`. Такой вызов создает и возвращает функцию, которая становится методом `deentityify`.

```
document.writeln(
    '&lt;&quot;&gt;'.deentityify());   // <">
```

Шаблон модуля использует возможности области видимости функции и возможности замыкания для представления функциональности и данных, связанных и упакованных вместе. В данном примере только метод `deentityify` имеет доступ к данным.

Обобщенный шаблон модуля — это функция, определяющая закрытые переменные и функции; создающая привилегированные функции, которые благодаря замыканию, получают доступ к закрытым переменным и функциям, и возвращающая привилегированные функции или хранящая их в доступном месте.

Программный шаблон модуля позволяет отказаться от использования глобальных переменных благодаря сокрытию информации и другим полезным наработкам. Этот подход очень эффективен в изолирующих приложениях.

Кроме того, он может пригодиться при создании защищенных объектов. Предположим, что необходимо создать объект, задающий серийный номер:

```
var serialjnaker = function () {

// Создает объект, задающий уникальные значения строк. Уникальная строка
// состоит из двух частей: префикса и порядкового номера. Для объекта
// определены методы задания префикса и порядкового номера,
// а также метод gensym, создающий уникальные строки.

    var prefix = '';
    var seq = 0;
```

```
    return {
        set_prefix: function (p) {
            prefix = String(p);
        },
        set_seq: function (s) {
            seq = s;
        },
        gensym: function ( ) {
            var result = prefix + seq;
            seq += 1;
            return result;
        }
    };
};

var seqer = serial_maker();
seqer.set_prefix('Q');
seqer.set_seq(1000);
var unique = seqer.gensym();    // unique принимает значение "Q1000"
```

Эти методы нельзя использовать с `this` или `that`, так как `seqer` подвергается риску. В этом случае невозможно будет получить или изменить `prefix` или `seq` без разрешенных методов. Объект `seqer` изменяемый, так что методы можно заменить, но это еще не дает полного доступа к нему. Метод `seqer` — это просто набор функций, которые дают возможность предоставлять определенные полномочия для получения или редактирования его скрытых состояний.

Если рассматривать функцию `seqer.gensym` с другой стороны, она должна иметь возможность создавать уникальные строки, не изменяя `prefix` или `seq`.

Каскады

Некоторые методы не имеют возвращаемого значения, что характерно, например, для методов, устанавливающих или изменяющих состояние объекта. В случае когда методы возвращают `this` вместо `undefined`, можно задействовать *каскады*. С помощью каскада можно последовательно вызвать несколько методов одного объекта в одном выражении. При наличии библиотеки Ajax, подключающей каскады, это могло бы выглядеть так:

```
getElement('myBoxDiv')
    .move(350, 150)
    .width(100)
    .height(100)
    .color('red')
    .border('10px outset')
    .padding('4px')
```

продолжение ⇩

```
    .appendText("Please stand by")
    .on('mousedown', function (m) {
        this.startDrag(m, this.getNinth(m));
    })
    .on('mousemove', 'drag')
    .on('mouseup', 'stopDrag')
    .later(2000, function () {
        this
            .color('yellow')
            .setHTML("What hath God wraught?")
            .slide(400, 40, 200, 200);
    })
        .tip("This box is resizeable.");
```

В этом примере функция `getElement` возвращает объект, предоставляющий функции для DOM-элемента `id="myBoxDiv"`. Эти методы позволяют удалить элемент, изменить его размер и стиль, добавить поведение. Каждый из этих методов возвращает объект, поэтому результат вызова может быть использован для следующего вызова.

Каскады позволяют строить очень выразительные интерфейсы, такие, которые дают возможность делать сразу много вещей.

Каррирование

Поскольку функции являются значениями, существует множество интересных способов ими манипулировать. *Каррирование*[1] позволяет создать новую функцию, объединяющую функцию и аргумент:

```
var add1 = add.curry(1);
document.writeln(add1(6));    // 7
```

Здесь `add1` — это функция, созданная путем передачи единицы методу `carry` объекта `add`. Функция `add1` добавляет единицу к своему аргументу. В JavaScript нет метода `carry`, но это можно исправить путем расширения объекта `Function.prototype`:

```
Function.method('curry', function ( ) {
    var args = arguments, that = this;
    return function ( ) {
        return that.apply(null, args.concat(arguments));
    };
});    // Что-то не так...
```

[1] *Каррирование* или *карринг* (англ. currying) — преобразование функции пары аргументов в функцию, берущую свои аргументы по одному. Это преобразование получило свое название в честь Х. Карри. — *Примеч. ред.*

Метод `carry` создает замыкание, которое хранит первоначальную функцию и аргументы в `carry`. Замыкание возвращает функцию, при вызове которой, в свою очередь, возвращается результат вызова первоначальной функции, в которую передаются все аргументы `carry` и текущего вызова. Для объединения двух массивов аргументов используется метод `concat`, принадлежащий `Array`.

К сожалению, как уже отмечалось, массив `arguments` не является массивом и поэтому не имеет метода `concat`. Для того чтобы избежать этого недостатка, можно применить метод массива `slice` к обоим массивам `arguments`. Этот метод создаст массивы, ведущие себя корректно при использовании метода `concat`:

```
Function.method('curry', function ( ) {
    var slice = Array.prototype.slice,
        args = slice.apply(arguments),
        that = this;
    return function ( ) {
        return that.apply(null, args.concat(slice.apply(arguments)));
    };
});
```

Мемоизация

Функции могут использовать объекты для хранения результатов предыдущих вычислений, что позволяет избежать ненужной работы. Для такой оптимизации, называемой *мемоизацией*, в JavaScript применяются объекты и массивы.

К примеру, пусть необходимо задать рекурсивную функцию для вычисления чисел Фибоначчи. Число Фибоначчи является суммой двух предыдущих чисел Фибоначчи. Первые два числа — это 0 и 1:

```
var fibonacci = function (n) {
    return n < 2 ? n : fibonacci(n - 1) + fibonacci(n - 2);
};

for (var i = 0; i <= 10; i += 1) {
    document.writeln('// ' + i + ': ' + fibonacci(i));
}

// 0: 0
// 1: 1
// 2: 1
// 3: 2
// 4: 3
// 5: 5
// 6: 8
// 7: 13
// 8: 21
// 9: 34
// 10: 55
```

Этот пример работает, но в нем много лишнего. Функция `fibonacci` вызывается 453 раза: 11 раз она вызывается в программе и еще 442 раза она вызывает себя для расчета значений, которые, вероятно, уже были вычислены. Если мемоизировать функцию, то можно значительно сократить рабочую нагрузку.

Будем хранить результаты мемоизации в массиве `memo`, который можно скрыть посредством замыкания. При вызове функция сначала проверяет, известен ли ей результат. Если да, то его следует немедленно вернуть:

```
var fibonacci = (function () {
    var memo = [0, 1];
    var fib = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = fib(n - 1) + fib(n - 2);
            memo[n] = result;
        }
        return result;
    };
    return fib;
}());
```

Эта функция возвращает тот же результат, но вызывается всего 29 раз: 11 раз она вызывается в программе и 18 раз она вызывает себя, чтобы получить ранее мемоизированные результаты.

Обобщить все сказанное можно, создав функцию мемоизации. Функция `memoizer` получает исходный массив `memo` и функцию `formula`. Она возвращает рекурсивную функцию, управляющую хранилищем `memo` и вызывающую функцию `formula` по мере необходимости. Функция `recur` вместе с параметрами передается в функцию `formula`:

```
var memoizer = function (memo, formula) {
    var recur = function (n) {
        var result = memo[n];
        if (typeof result !== 'number') {
            result = formula(recur, n);
            memo[n] = result;
        }
        return result;
    };
    return recur;
};
```

Теперь `fibonacci` объявляется с помощью функции `memoizer`, получающей исходный массив `memo` и функцию `formula`:

```
var fibonacci = memoizer([0, 1], function (recur, n) {
    return recur(n - 1) + recur(n - 2);
});
```

Разрабатывая функции, создающие другие функции, можно значительно сократить итоговый объем работы. Например, для создания функции мемоизации факториала нужно всего лишь написать основную формулу факториала:

```
var factorial = memoizer([1, 1], function (recur, n) {
    return n * recur(n - 1);
});
```

5 Наследование

Дробится вещь на множество частей.
Картины есть такие: если взглянешь
На них вблизи, то видишь только пятна,

Уильям Шекспир. Ричард II

Наследование — весьма важный раздел большинства языков программирования.

В классических языках (например, Java) наследование (или расширение) дает две довольно полезные возможности. Во-первых, это способ многократного использования кода. Если новый класс похож на существующий, остается только описать различия. Готовые программные шаблоны могут существенно снизить затраты на разработку программного обеспечения. Еще одним преимуществом классического наследования является то, что оно включает в себя спецификацию системы типов. Это освобождает программиста от необходимости описывать операции для явного приведения типов, что, вообще говоря, очень даже неплохо, так как приведение типов может поставить под угрозу безопасность системы типов.

В JavaScript, где нет типизации, не существует и приведения типов. Происхождение объекта не имеет значения. Важно знать, на что объект способен, а не то, от кого он происходит.

JavaScript предоставляет богатый набор готовых программных шаблонов для многократного использования кода. JavaScript позволяет сымитировать классический шаблон, но также предлагает и другие шаблоны, часто более выразительные. Набор доступных программных шаблонов наследования в JavaScript огромен. В этой главе подробно рассмотрены только некоторые из самых простых. Можно создавать и гораздо более сложные конструкции, но, как правило, чем проще, тем лучше.

В классических языках объекты являются экземплярами классов, а каждый класс может наследоваться от другого класса. JavaScript — это язык прототипов, а это означает, объекты наследуются непосредственно от других объектов.

Псевдоклассовое наследование

JavaScript противоречит своей прототипизированной природе. Механизм прототипов скрыт за сложными синтаксическими правилами, смутно напоминающими классические. Вместо того чтобы наследоваться непосредственно от других объектов, объекты создаются с помощью конструктора функций.

При создании объекта функции конструктор `Function` работает примерно так:

```
this.prototype = {constructor: this};
```

Новый объект функции задается свойством `prototype`, причем его значением является объект, содержащий свойство `constructor`, значением которого, в свою очередь, является новый объект функции. Объект `prototype` — это то место, где хранятся унаследованные черты. Каждая функция получает объект `prototype`, потому что в JavaScript нет возможности определить, какие функции предназначены для использования в качестве конструктора. Свойство `constructor` не так важно, как объект `prototype`.

Если функция вызывается посредством шаблона вызова конструктора с префиксом `new`, меняется способ исполнения функции. Если бы оператор `new` был методом, а не оператором, функция могла бы быть реализована следующим образом:

```
Function.method('new', function () {

// Создаем новый объект, который наследуется от конструктора prototype.

    var that = Object.create(this.prototype);

// Вызов конструктора, связывающего this с новым объектом.

    var other = this.apply(that, arguments);

// Если возвращаемое значение не является объектом,
// оно заменяется новым объектом.

    return (typeof other === 'object' && other) || that;
});
```

Можно определить конструктор и расширить его прототип:

```
var Mammal = function (name) {
    this.name = name;
};

Mammal.prototype.get_name = function () {
    return this.name;
};
```

продолжение ⇩

```
Mammal.prototype.says = function () {
    return this.saying || '';
};
```

Теперь можно сделать, например, следующее:

```
var myMammal = new Mammal('Herb the Mammal');
var name = myMammal.get_name(); // 'Herb the Mammal'
```

Можно создать еще один псевдокласс, наследуемый от `Mammal`, определив собственную функцию `constructor` и заменить ее свойство `prototype` экземпляром `Mammal`:

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
};

// Заменяем Cat.prototype новым экземпляром Mammal

Cat.prototype = new Mammal();

// Расширяем новый с методами purr and get_name.

Cat.prototype.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s +=
        }
        s += 'r';
    }
    return s;
};
Cat.prototype.get_name = function () {
    return this.says( ) + ' ' + this.name + ' ' + this.says( );
};

var myCat = new Cat('Henrietta');
var says = myCat.says(); // 'meow'
var purr = myCat.purr(5); // 'r-r-r-r-r'
var name = myCat.get_name();
// 'meow Henrietta meow'
```

Программный шаблон псевдокласса должен напоминать объектно-ориентированный класс, но выглядит он иначе. Некоторые уродства можно скрыть с помощью метода `method` и определения метода `inherits`:

```
Function.method('inherits', function (Parent) {
    this.prototype = new Parent(); return this;
});
```

Методы `inherits` и `method` возвращают `this`, что позволяет использовать каскады. Теперь можно задать `Cat` одной инструкцией:

```
var Cat = function (name) {
    this.name = name;
    this.saying = 'meow';
}.
    inherits(Mammal).
    method('purr', function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s +=
            }
            s += 'r';
        }
        return s;
    }).
    method('get_name', function ( ) {
        return this.says( ) + ' ' + this.name + ' ' + this.says( );
    });
```

Теперь, благодаря тому, что наиболее броские черты `prototype` скрыты, наследование выглядит чуть менее непривычным. Но разве ситуация действительно улучшилась? Появились конструкторы функций, действующие как классы, но имеющие странное поведение. И никакой приватности — все свойства открыты.

Хуже того, существует серьезная опасность, связанная с использованием конструктора функций. Если при вызове конструктора функций отсутствует префикс `new`, то `this` не связывается с новым объектом. К сожалению, `this` связывается с глобальным объектом, так что вместо создания нового объекта произойдет замена глобальных переменных. Это действительно ужасно. Причем предупреждение не возникнет ни на этапе компиляции, ни при выполнении программы.

Это серьезная ошибка разработчиков языка. Чтобы избежать этой проблемы, принято соглашение, согласно которому все названия конструкторов функций пишутся с заглавной буквы и далее нигде в названии заглавная буква не используется. Остается надеяться, что при просмотре удастся найти недостающие префиксы `new`. Однако гораздо лучше вовсе не использовать оператор `new`.

Хотя псевдоклассовая форма наследования удобна для программистов, не знакомых с JavaScript, она скрывает истинную природу языка. Схожесть с классическими языками может заставить программистов создавать излишне глубокие и сложные иерархии. Сложные иерархии классов, как правило, связаны с ограничениями в плане проверки статических типов. Язык JavaScript лишен подобных ограничений. В классических языках наследование классов является единственной формой многократного использования кода, в то время как JavaScript предоставляет для этого гораздо больше возможностей.

Спецификаторы объектов

Бывает, что в конструктор передается довольно большое число параметров. Это может вызвать уйму хлопот, поскольку очень трудно запомнить порядок следования аргументов. В таких случаях гораздо удобнее написать конструктор, принимающий один спецификатор объекта, содержащий спецификацию конструируемого объекта. Таким образом:

```
var myObject = maker(f, l, m, c, s);
```

Вместо этой инструкции можно написать следующее:

```
var myObject = maker({
    first: f,
    last: l,
    middle: m,
    state: s,
    city: c
});
```

Теперь аргументы можно перечислять в любом порядке, при использовании конструктора по умолчанию их можно вообще опустить, что делает код более понятным.

Это второе преимущество при работе с JSON (детали см. в приложении Д). С помощью JSON-текста можно описывать только данные, но иногда данные представляют собой объекты, поэтому было бы полезно связывать данные с их методами. Решить эту проблему не сложно, если конструктор будет принимать спецификатор объекта, так как достаточно просто передать JSON-объект в конструктор, который вернет полностью сформированный объект.

Прототипизированное наследование

В чисто прототипизированных программных шаблонах можно обойтись без классов. Вместо этого необходимо сосредоточиться на объектах. Такое наследование концептуально проще, чем классическое: новый объект может наследовать свойства старого объекта. Это не очень привычно, но вполне понятно. Сначала можно сделать какие-то полезные объекты, а затем уже на их основе создать множество похожих объектов. Классификации, то есть процесса разбиения приложений на множество вложенных абстрактных классов, можно полностью избежать.

Начнем с создания объекта с помощью литерала объекта:

```
var myMammal = {
    name : 'Herb the Mammal',
    get_name : function ( ) {
        return this.name;
    },
```

```
    says : function ( ) {
        return this.saying || '';
    }
};
```

Как только удается получить нужный объект, с помощью `Object` можно создавать экземпляры этого объекта. Для этого можно воспользоваться методом `create` из главы 3, а затем настроить каждый из новых экземпляров:

```
var myCat = Object.create(myMammal);
myCat.name = 'Henrietta';
myCat.saying = 'meow';
myCat.purr = function (n) {
    var i, s = '';
    for (i = 0; i < n; i += 1) {
        if (s) {
            s +=
        }
        s += 'r';
    }
    return s;
};
myCat.get_name = function () {
    return this.says( ) + ' ' + this.name + ' ' + this.says( );
};
```

Это пример *дифференцированного наследования*, основанного на изменении нового объекта с указанием отличий от объекта-родителя.

Иногда это может быть полезно при наследовании структур данных. Например, пусть необходимо сделать грамматический разбор фрагмента внутри фигурных скобок, написанного на JavaScript или TEX. Элементы, определенные внутри скобок, не видны за их пределами. В некотором смысле, область внутри скобок наследуется от внешней области. В JavaScript такое отношение можно легко представить с помощью объектов. Когда встречается открывающая фигурная скобка, вызывается тело функции. Функция `parse` ищет символы внутри `scope` и расширяет область видимости при обнаружении новых символов:

```
var block = function () {

// Запоминаем текущее значение scope. Создаем новый объект scope,
// включающий в себя все значения scope, начиная с текущего.
    var oldScope = scope;
    scope = Object.create(scope);

// Advance получает открывающую фигурную скобку.
    advance('{');

// Parse использует новое значение scope.
    parse(scope);
```

продолжение ⇩

```
// Advance получает закрывающую фигурную скобку и отбрасывает
// новое значение scope, возвращая старое.
    advance('}');
    scope = oldScope;
};
```

Функциональное наследование

Одним из недостатков программных шаблонов наследования, как уже отмечалось, является отсутствие возможности добиться приватности. Все свойства объекта оказываются видимыми, приватных переменных и методов не существует. Иногда это совсем не важно, но бывают ситуации, когда приватность имеет огромное значение. К сожалению, некоторые плохо осведомленные программисты принимают шаблоны за механизм, обеспечивающий приватность. Если необходимо задать приватное свойство, они дают ему странное название в надежде, что другие программисты, использующие этот код, сделают вид, что не видят этих странных членов объекта. К счастью, есть прекрасная альтернатива этому — программный шаблон модуля.

Для начала необходимо создать функцию, производящую объекты. Дадим ей имя, начинающееся со строчной буквы, поскольку нет необходимости использовать префикс `new`. Функция выполняется в четыре этапа:

1. Создает новый объект. Существует множество способов создать объект: с помощью литерала объекта, путем вызова конструктора функции с префиксом `new`, методом `Object.create`, который создает новый экземпляр уже существующего объекта, или вызвав любую функцию, возвращающую объект.
2. При необходимости определяет приватные методы и переменные экземпляра. Это обычные переменные функции.
3. Дополняет новый объект методами, которые будут иметь привилегированный доступ к параметрам и переменным, определенным на втором этапе.
4. Возвращает новый объект.

Вот псевдокод программного шаблона функционального конструктора (полужирным шрифтом выделены основные этапы создания функции):

```
var constructor = function (spec, my) {
    var that, другие приватные переменные;
    my = my || {};

    Дополняем my открытыми переменными и функциями

    that = a new object;

    Дополняем that привилегированными методами

    return that;
};
```

Объект `spec` содержит всю информацию, необходимую конструктору для создания экземпляра. Эти данные могут быть скопированы в приватные переменные или преобразованы другими функциями. Кроме того, в случае необходимости методы могут получить доступ к данным из `spec` (для упрощения можно заменить `spec` одним значением, что бывает полезно, когда при создании объекта не требуется весь объект `spec`).

Объект `my` представляет собой контейнер со скрытыми данными, доступными для конструкторов в цепи наследования. Совершенно не обязательно использовать объект `my`. Если `my` не будет передан конструктору, тот создаст объект `my` сам.

Несколько слов об объявлении приватных методов и переменных экземпляра объекта. Достаточно просто объявить переменные. Переменные и внутренние функции конструктора будут приватными членами экземпляра. Внутренние функции будут иметь доступ к `spec`, `my`, `that` и приватным переменным.

Затем путем присваивания добавим объекту `my` скрытые данные:

```
my.member = value;
```

Теперь создадим новый объект и свяжем его с `that`. Существует множество способов создать объект: с помощью литерала объекта, путем вызова псевдоклассового конструктора функции с префиксом `new`, с использованием метода `Object.create` для прототипа объекта или вызвав конструктор функции и передав ему объект `spec` (возможно, тот же объект `spec`, который был передан при создании самого конструктора) и объект `my`. Объект `my` предоставляет другому конструктору доступ к вложенным данным. Другой конструктор также может предоставить свои скрытые данные `my`, которыми может воспользоваться конструктор объекта `my`.

Кроме того, можно расширить `that`, добавив привилегированные методы, составляющие интерфейс объекта. Можно задать новые функции, которые будут являться членами `that`. Хотя безопаснее сначала определить функции в качестве приватных методов, а затем связать их с `that`:

```
var methodical = function () {
    ...
};
that.methodical = methodical;
```

Преимущество определения `methodical` в два этапа состоит в следующем: если понадобится вызвать `methodical` для другого метода, можно будет вызвать `methodical()` вместо `that.methodical()`. Если экземпляр был поврежден или изменен таким образом, что метод `that.methodical` оказался замененным методом, вызывающим `methodical`, он продолжит работать, как раньше, поскольку приватная функция `methodical` не влияет на изменение экземпляра.

Ну и в конце возвращаем `that`.

Давайте исследуем этот программный шаблон на примере с млекопитающими. Здесь нет необходимости использовать объект `my`, поэтому его можно просто отбросить, но объект `spec` понадобится.

Свойства `name` и `saying` сейчас полностью закрыты. Они доступны только через привилегированные методы `get_Name` и `says`:

```
var mammal = function (spec) {
    var that = {};

    that.get_name = function () {
        return spec.name;
    };

    that.says = function () {
        return spec.saying || '';
    };

    return that;
};

var myMammal = mammal({name: 'Herb'});
```

В псевдоклассовом программном шаблоне функция конструктора `Cat` дублировала работу конструктора `Mammal`. Это совершенно не обязательно в функциональном шаблоне, потому что конструктор `Cat` вызовет конструктор `Mammal`, позволяя `Mammal` выполнить большую часть работы по созданию объекта, так что конструктору `Cat` останется только позаботиться о различиях:

```
var cat = function (spec) {
    spec.saying = spec.saying || 'meow';
    var that = mammal(spec);
    that.purr = function (n) {
        var i, s = '';
        for (i = 0; i < n; i += 1) {
            if (s) {
                s +=
            }
            s += 'r';
        }
        return s;
    };
    that.get_name = function () {
        return that.says() + ' ' + spec.name + ' ' + that.says();
    };
    return that;
};

var myCat = cat({name: 'Henrietta'});
```

Использование функционального программного шаблона позволяет также иметь дело с суперметодами. Создадим метод `superior`, получающий имя метода и возвращающий функцию, которая вызывает этот метод. Функция будет вызывать оригинальный метод, даже если свойство изменится:

```
Object.method('superior', function (name) {
    var that = this,
        method = that[name];
    return function ( ) {
        return method.apply(that, arguments);
    };
});
```

Испробуем этот подход на объекте `coolcat`, похожем на `Cat`, но имеющем собственный метод `get_Name`, который вызывает суперметод. Потребуется небольшая подготовка. Объявим переменную `super_get_name` и присвоим ей результат вызова метода `superior`:

```
var coolcat = function (spec) {
    var that = cat(spec),
        super_get_name = that.superior('get_name');
    that.get_name = function (n) {
        return 'like ' + super_get_name() + ' baby';
    };
    return that;
};

var myCoolCat = coolcat({name: 'Bix'});
var name = myCoolCat.get_name();
// 'like meow Bix meow baby'
```

Функциональный программный шаблон довольно гибок. Его применение требует меньше усилий, чем применение псевдоклассового шаблона, и при этом лучше обеспечивает инкапсуляцию и скрытие информации, а также доступ к суперметодам.

Объект защищен от взлома, если все его свойства будут приватными. Свойства объекта могут быть заменены или удалены, но целостность объекта не будет нарушена. Если создать объект согласно функциональному шаблону, и его методы не будут использовать `this` или `that`, то объект будет надежен. Надежный объект — это просто набор функций, предоставляющих определенные возможности.

Надежный объект не может находиться под угрозой. Доступ к надежным объектам не даст злоумышленнику возможности получить доступ к внутреннему состоянию объекта, кроме как посредством разрешенных методов.

Детали

Можно создавать объекты из множества деталей. Например, создать функцию, добавляющую простые функции обработки событий к любому объекту. Эта функция добавляет методы `on`, `fire` и приватные события реестра:

```
var eventuality = function (that) {
    var registry = {};
```

продолжение ⇩

```
    that.fire = function (event) {

// Fire - событие объекта on. Событием может быть либо строка, содержащая
// название события, либо объект, который содержит свойство type,
// хранящее имя события. Обработчики регистрируются методом 'on',
// который вызывается в соответствии с именем события.

        var array,
            func,
            handler,
            i,
            type = typeof event === 'string' ? event : event.type;
// Если для этого события существует массив обработчиков,
// то пройдем по нему и выполним все обработчики по порядку.

        if (registry.hasOwnProperty(type)) {
            array = registry[type];
            for (i = 0; i < array.length; i += 1) {
                handler = array[i];

// Запись обработчика содержит метод и массив параметров (не обязательно).
// Если метод - это имя, найдем функцию.

                func = handler.method;
                if (typeof func === 'string') {
                    func = this[func];
                }

// Вызваем обработчик. Если запись содержит параметры, передаем их.
// В противном случае переходим к объекту события.

                func.apply(this,
                    handler.parameters || [event]);
            }
        }
        return this;
    };

    that.on = function (type, method, parameters) {

// Региструем событие. Создаем обработчик записи. Помещаем его
// в массив обработчиков, создавая, если его еще не существует
// для этого значения type.

        var handler = {
            method: method,
            parameters: parameters
        };
```

```
        if (registry.hasOwnProperty(type)) {
            registry[type].push(handler);
        } else {
            registry[type] = [handler];
        }
        return this;
    };
    return that;
};
```

Можно вызвать `eventuality` для любого объекта, наделяя его методами обработки событий. Также можно вызвать его внутри функции конструктора до того, как возвращен объект `that`:

```
eventuality(that);
```

Таким образом, конструктор может собирать объекты из набора деталей. Свободная типизация в JavaScript здесь является огромным преимуществом, поскольку отсутствует система типов, для которой важно происхождение классов. Вместо этого можно сосредоточиться на характере их содержимого.

Если необходимо дать функции `eventuality` доступ к приватным состояниям объекта, можно передать ей объект `my`.

6 Массивы

Ты, волк в овечьей шкуре, с глаз долой!

Уильям Шекспир. Генрих VI. Часть 1

Массив представляет собой непрерывный фрагмент памяти, доступ к элементам которого осуществляется с помощью целых чисел (индексов), использующихся для вычисления смещения. Массивы могут обеспечивать быстрый доступ к данным. К сожалению, в JavaScript нет ничего похожего на такое определение массива.

Однако в JavaScript есть объекты, чем-то напоминающие массивы. Индексы массива преобразуются в строки, которые используются для создания свойств объекта. Хотя такие структуры значительно медленнее реальных массивов, они могут оказаться удобнее в использовании. Получение и изменение свойств происходит так же, как в случае объектов, исключая трюк с именами целочисленных свойств. Массивы имеют собственный формат литерала. Кроме того, существует довольно большой набор полезных встроенных методов массивов, которые описаны в главе 8.

Литералы массивов

С помощью литералов массивов довольно удобно задавать новые значения массива. Литерал массива представляет собой пару квадратных скобок, которая содержит несколько значений, разделенных запятыми, или не содержит вообще ни одного значения. Литерал массива может указываться в любом выражении. Первое значение внутри скобок получает имя свойства `'0'`, второе — имя свойства `'1'` и т. д.:

```
var empty = [];
var numbers = [
    'zero', 'one', 'two', 'three', 'four',
    'five', 'six', 'seven', 'eight', 'nine'
];
```

```
empty[1]          // undefined
numbers[1]        // 'one'

empty.length      // 0
numbers.length    // 10
```

Литерал объекта:

```
var numbers_object = {
    '0': 'zero', '1': 'one', '2': 'two',
    '3': 'three', '4': 'four', '5': 'five',
    '6': 'six', '7': 'seven', '8': 'eight',
    '9': 'nine'
};
```

Результат в обоих случаях аналогичный. Массив `numbers`, как и `numbers_object`, является объектом, содержащим 10 свойств, имеющих точно такие же имена и значения. Но есть и существенные различия. Объект `numbers` наследуется от `Array.prototype`, в то время как `numbers_object` — от `Object.prototype`, поэтому `numbers` наследует больший набор полезных методов. Кроме того, `numbers` получает скрытое свойство `length`, а `numbers_object` нет.

В большинстве языков все элементы массива должны быть одинакового типа. JavaScript позволяет создавать массивы, содержащие самые разные значения:

```
var misc = [
    'string', 98.6, true, false, null, undefined,
    ['nested', 'array'], {object: true}, NaN,
    Infinity
];
misc.length        // 10
```

Длина

Каждый массив имеет свойство `length`. В отличие от большинства других языков, длина массива `length` в JavaScript не является его верхней границей. Если необходимо хранить элемент с индексом, большим или равным текущему значению `length`, то оно увеличится до требуемого значения. Благодаря этому не возникнет ошибки переполнения массива.

Значение свойства `length` равно наибольшему индексу в массиве плюс один. Это значение может не соответствовать количеству свойств в массиве:

```
var myArray = [];
myArray.length    // 0

myArray[1000000] = true;
myArray.length    // 1000001
// myArray содержит одно свойство.
```

Постфиксный оператор индексирования [ ] преобразует выражение в строку, используя для выражения метод toString, если он существует. Эта строка становится именем свойства. Если строка выглядит как натуральное число, большее или равное текущему значению длины массива и меньшее 4294 967 295, то значение length устанавливается на единицу большим нового индекса.

Значение length можно задать в явном виде. Если задать его больше, чем потребуется для массива, память под незаданные значения на выделяется. Однако если задать значение length меньше, чем необходимо, то все свойства с индексом, большим или равным новой длине массива, удаляются:

```
numbers.length = 3;
// numbers имеет значения ['zero', 'one', 'two']
```

Если присвоить новому элементу индекс, который равен текущему значению length, то элемент добавляется в конец массива:

```
numbers[numbers.length] = 'shi';
// numbers имеет значения ['zero', 'one', 'two', 'shi']
```

Иногда для этого удобнее пользоваться методом push:

```
numbers.push('go');
// numbers имеет значения ['zero', 'one', 'two', 'shi', 'go']
```

Удаление

Так как массивы в JavaScript на самом деле являются объектами, для удаления элементов из массива может быть использован оператор delete:

```
delete numbers[2];
// numbers имеет значения ['zero', 'one', undefined, 'shi', 'go']
```

Такой способ, к сожалению, оставляет «дырки» в массиве, поскольку у элементов, расположенных правее удаляемого, остаются исходные имена. Для того чтобы добиться желаемого результата, требуется поочередно сдвинуть все последующие элементы справа налево.

Однако для работы с массивами в JavaScript существует также метод splice, позволяющий удалить несколько элементов массива и заменить их другими. Первый аргумент — это порядковый номер элемента, с которого требуется начать удаление из массива. Второй аргумент — число элементов для удаления. Все дополнительные аргументы вставляются в указанную первым аргументом точку массива:

```
numbers.splice(2, 1);
// numbers имеет значения ['zero', 'one', 'shi', 'go']
```

Здесь ключ свойства, имеющего значение 'shi', меняется с '3' на '2'. Так как для каждого свойства, следующего за удаленным, ключ должен быть уничтожен

и заменен новым, это довольно длительный процесс, особенно для больших массивов.

Перечисление

Благодаря тому, что в JavaScript массивы — это объекты, для перемещения по элементам массива может использоваться инструкция `for in`. К сожалению, `for in` не дает никаких гарантий относительно порядка предоставления элементов, в то время как для большинства приложений требуется получать элементы массива поочередно (согласно их индексу). Кроме того, существует проблема с необычными свойствами, полученными по цепочке прототипов.

К счастью, обычная инструкция `for` позволяет избежать этих проблем. Инструкция `for` в JavaScript выполняется примерно так же, как в большинстве C-подобных языков. Ее выполнение зависит от трех вещей: начала цикла, условия выполнения цикла и шага цикла.

```
var i;
for (i = 0; i < myArray.length; i += 1) {
    document.writeln(myArray[i]);
}
```

Путаница

Распространенной ошибкой при написании программ на JavaScript является использование объекта, когда требуется массив, или массива, когда требуется объект. Правило простое: если имена свойств — это небольшая последовательность целых чисел, следует использовать массив. В противном случае — объект.

Язык JavaScript и сам порождает путаницу между массивами и объектами. Оператор `typeof` в качестве типа массива возвращает `'object'`, что совсем не способствует прояснению ситуации.

В JavaScript нет хорошего механизма, который позволили бы различать массивы и объекты. Обойти этот недостаток можно, определив собственную функцию `is_array`:

```
var is_array = function (value) {
    return value && typeof value === 'object' &&
    value.constructor === Array;
};
```

К сожалению, она не сможет идентифицировать массивы, создаваемые в другом окне или фрейме. Если необходимо точно идентифицировать такие внешние массивы, функцию `is_array` придется немного усложнить:

```
var is_array = function (value) {
    return Object.prototype.toString.apply(value) === '[object Array]';
};
```

Методы

JavaScript предоставляет ряд методов для работы с массивами. Эти методы являются функциями, хранящимися в `Array.prototype`. Как отмечено в главе 3, объект `Object.prototype` может быть расширен. Также может быть расширен и `Array.prototype`.

Предположим, что требуется добавить в массив метод, позволяющий производить вычисления с элементами массива:

```
Array.method('reduce', function (f, value) {
    var i;
    for (i = 0; i < this.length; i += 1) {
        value = f(this[i], value);
    }
    return value;
});
```

Каждый массив наследует метод, созданный при добавлении функции в `Array.prototype`. В данном примере был определен метод `reduce`, который принимает функцию и начальное значение `value` и для каждого элемента массива вызывает функцию, аргументами которой являются элемент массива и переданное начальное значение, а затем вычисляет новое значение переменной `value`. Когда цикл заканчивается, функция возвращает переменную `value`. Если передать функцию, которая складывает два числа, метод вычислит сумму. Если передать функцию, которая перемножает два числа, — произведение:

```
// Создаем массив чисел.

var data = [4, 8, 15, 16, 23, 42];

// Объявляем две простые функции. Одна будет суммировать
// два числа. Другая будет перемножать два числа.

var add = function (a, b) {
    return a + b;
};

var mult = function (a, b) {
    return a * b;
};
```

```
// Вызываем метод reduce для массива data, передавая
// ему функцию сложения.

var sum = data.reduce(add, 0);    // sum имеет значение 108

// Снова вызываем метод reduce, на этот раз передаем
// ему функцию умножения.

var product = data.reduce(mult, 1);
    // product имеет значение 7418880
```

Поскольку массив на самом деле является объектом, можно добавлять методы непосредственно для каждого массива:

```
// Задаем функцию total для массива data.

data.total = function () {
    return this.reduce(add, 0);
};

total = data.total();    // total имеет значение 108
```

Так как строка `'total'` не является целым числом, добавление в массив свойства `total` не приведет к изменению значения `length`. Удобство использования массивов в том, что хотя имена свойств и являются целыми числами, массивы попрежнему остаются объектами, а объекты могут принимать любую строку в качестве имени свойства.

Нет смысла использовать с массивами метод `Object.create` из главы 3, поскольку он создает объект, а не массив. Созданный таким образом объект будет наследовать значения и методы массива, но не будет иметь свойства `length`.

Размерность

В JavaScript массивы обычно не инициализируются. Если создать новый массив с помощью оператора `[ ]`, то он будет пустым. Если обратиться к недостающим элементам, будет возвращено значение `undefined`. Если осознавать это или задавать каждый элемент прежде, чем пытаться его получить, то все будет хорошо. Однако если реализовать алгоритмы в предположении, что каждый элемент начинается с определенного значения (такого как 0), следует подготовить массив самостоятельно. Язык JavaScript мог бы в той или иной степени обеспечить `Array.dim` средствами делать нечто подобное, но и самостоятельно можно легко исправить это упущение:

```
Array.dim = function (dimension, initial) {
    var a = [], i;
    for (i = 0; i < dimension; i += 1) {
```

продолжение ⇩

```
        a[i] = initial;
    }
    return a;
};

// Создадим массив, состоящий из 10 нулей.

var myArray = Array.dim(10, 0);
```

В JavaScript нет массивов, имеющих размерность больше единицы, но, как и в большинстве C-подобных языков, можно создавать массивы массивов:

```
var matrix = [
    [0, 1, 2],
    [3, 4, 5],
    [6, 7, 8]
];
matrix[2][1]     // 7
```

Для того чтобы создать двухмерный массив или массив массивов, необходимо объявить собственный массив:

```
for (i = 0; i < n; i += 1) {
    my_array[i] = [];
}

// Примечание: в данной ситуации Array.dim (n, []) не сработает.
// Каждый элемент получит ссылку на один и тот же
// массив, что плохо.
```

Первоначально ячейки пустой матрицы будут иметь значение `undefined`. Если возникнет необходимость, чтобы они имели те или иные начальные значения, придется задать их явно. Опять же, язык JavaScript мог бы предоставить более качественную поддержку матриц. Однако и этот недостаток можно исправить:

```
Array.matrix = function (m, n, initial) {
    var a, i, j, mat = [];
    for (i = 0; i < m; i += 1) {
        a = [];
        for (j = 0; j < n; j += 1) {
            a[j] = initial;
        }
        mat[i] = a;
    }
    return mat;
};

// Создадим матрицу 4 * 4, заполненную нулями.
```

```
var myMatrix = Array.matrix(4, 4, 0);

document.writeln(myMatrix[3][3]);    // 0

// Создадим для матрицы метод identity.

Array.identity = function (n) {
    var i, mat = Array.matrix(n, n, 0);
    for (i = 0; i < n; i += 1) {
        mat[i][i] = 1;
    }
    return mat;
};

myMatrix = Array.identity(4);

document.writeln(myMatrix[3][3]);    // 1
```

7 Регулярные выражения

Брак по любви блаженством наделяет
И образ мира высшего являет.
На ком жениться должен наш король...

Уильям Шекспир. Генрих VI. Часть 1

Многие черты JavaScript позаимствованы у других языков. Синтаксис взят у Java, функции — у Scheme, прототипизированное наследование — у Self. Ну а механизм работы с регулярными выражениями JavaScript позаимствовал у Perl.

Регулярное выражение — это спецификация синтаксиса простого языка. Регулярные выражения необходимы для реализации методов поиска, замены и получения информации из строк. К методам для работы с регулярными выражениями относятся `regexp.exec`, `regexp.test`, `string.match`, `string.replace`, `string.search` и `string.split` — все они подробно описаны в главе 8. В JavaScript регулярные выражения, как правило, значительно производительнее эквивалентных строковых операций.

Регулярные выражения основаны на математической теории формализованных языков. Кен Томпсон (Ken Thompson) адаптировал теоретическую работу Стивена Клини (Stephen Kleene) о языках типа 3 в реальном шаблоне соответствия, который можно встраивать в программные продукты, такие как текстовые редакторы и языки программирования.

Синтаксис регулярных выражений в JavaScript почти полностью соответствует оригинальной формулировке из Bell Labs, за исключением нескольких переосмысленных фрагментов и расширений, заимствованных из Perl. Правила написания регулярных выражений могут быть на удивление сложными, поскольку одни и те же символы в разных позициях могут интерпретироваться и как операторы, и как литералы. А еще хуже то, что чем сложнее написано регулярное выражение, тем труднее в нем разобраться и тем опаснее его модифицировать. Нужно иметь достаточно полное представление о сложности регулярных выражений, чтобы правильно их читать. Чтобы облегчить себе задачу, немного упростим правила. Представляемые здесь регулярные выражения немного компактнее и лаконичнее,

поэтому их будет проще правильно использовать. И это хорошо, потому что регулярные выражения часто вызывают затруднения в поддержке и отладке.

Сегодня использование регулярных выражений не является обязательным, но может быть довольно полезным. Регулярные выражения, как правило, настолько лаконичны, что напоминают шифр. Их легко использовать в простейшей форме, но при этом они быстро могут завести вас в тупик. Читать регулярные выражения в JavaScript может быть трудно отчасти еще и потому, что в них не может быть ни комментариев, ни пробелов. Все части регулярного выражения следуют друг за другом, что делает его почти нечитабельным. Это обстоятельство вызывает особую обеспокоенность, если регулярные выражения используются в приложениях для поиска и проверки. Если невозможно прочитать и понять регулярное выражение, как можно быть уверенным, что оно будет работать правильно? Тем не менее, несмотря на их очевидные недостатки, регулярные выражения широко используются.

Пример

А вот и пример. Это регулярное выражение, соответствующее URL-адресу. Страницы этой книги не бесконечно широкие, поэтому разобьем его на две строки. В JavaScript-коде регулярное выражение должно представлять собой одну строку, так как пробелы имеют значение:

```
var parse_url = /A(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)
(?::(\d+))?(?:\/([A?#]*))?(?:\?([A#]*))?(?:#(.*))?$/;

var url = "http://www.ora.com:80/goodparts?q#fragment";
```

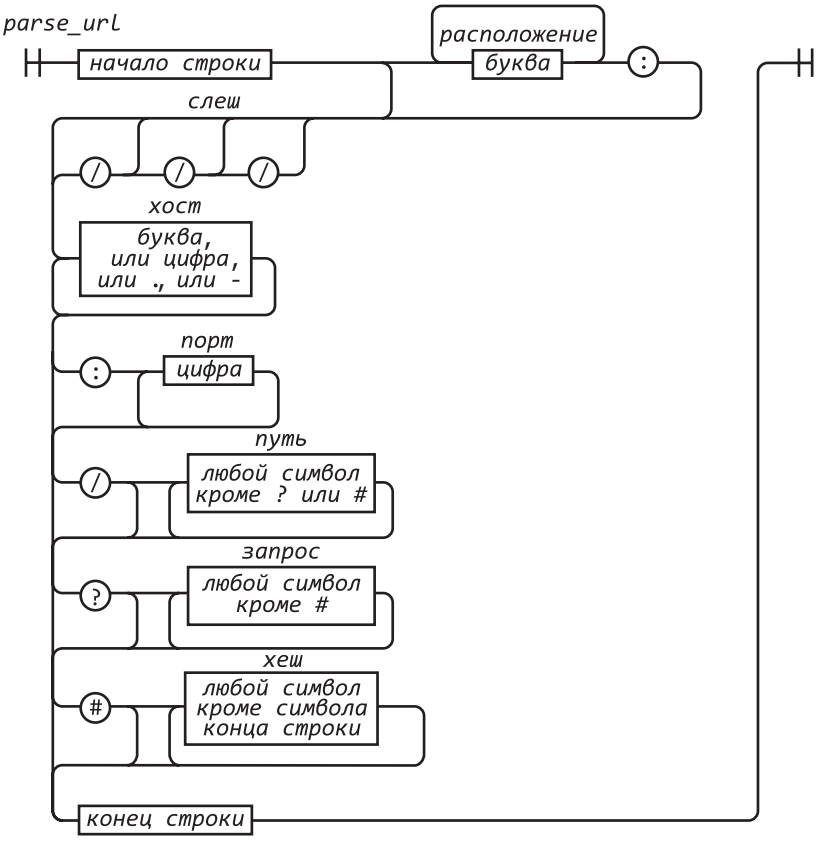
Вызовем метод `exec` для `parse_url`. Если регулярное выражение `parse_url` соответствует переданной методу строке, то он вернет массив, содержащий части, извлеченные из URL-адреса:

```
var url = "http://www.ora.com:80/goodparts?q#fragment";

var result = parse_url.exec(url);

var names = ['url', 'scheme', 'slash', 'host', 'port',
    'path', 'query', 'hash'];

var blanks = '    ';
var i;

for (i = 0; i < names.length; i += 1) {
    document.writeln(names[i] + ':' +
        blanks.substring(names[i].length), result[i]);
}
```

В результате получаем:

```
url:    http://www.ora.com:80/goodparts?q#fragment scheme: http
slash:  //
host:   www.ora.com
port:   80
path:   goodparts
query:  q
hash:   fragment
```

В главе 2 для описания языка JavaScript использовались синтаксические диаграммы. С помощью аналогичных диаграмм можно описать язык, заданный регулярными выражениями. Благодаря синтаксическим диаграммам можно легко увидеть, что делает регулярное выражение. Вот диаграмма `parse_url`:



Регулярные выражения не могут быть разбиты на более мелкие фрагменты подобно функциям, поэтому требуется время, чтобы отследить выполнение выражения `parse_url`.

Для того чтобы увидеть, как оно работает, рассмотрим выражение `parse_url` по частям.

```
^
```

Знак ^ указывает на начало строки — это якорь, который не дает при выполнении пропустить префикс, не свойственный для URL-адресов.

```
(?:([A-Za-Z]+):)?
```

Этот фрагмент задает расположение, но только в том случае, если за ним следует двоеточие (:). Знаки (?:...) обозначают не захватываемую группу. Суффикс ? показывает, что группа не является обязательной.

Это значит, что такая группа ни разу не повторяется или повторяется только один раз. Знаки (...) обозначают захватываемую группу. Захватываемая группа копирует соответствия, найденные в тексте, и помещает их в результирующий массив. Каждой захватываемой группе присваивается номер. Номер первой захватываемой группы — 1, поэтому совпадения, найденные с помощью этой группы, в результирующем массиве будут иметь индекс [1]. Класс символов обозначается знаками [...]. Класс символов A-Za-z содержит 26 прописных и 26 строчных букв. Диапазон от A до Z задается дефисом. Суффикс + означает, что класс символов может иметь одно и более соответствие. Группа, за которой следует двоеточие (:), — это символ, которому необходимо найти точное соответствие.

```
(\/{0,3})
```

Следующий фрагмент — это захватываемая группа номер 2. Знаки \/ указывают, что необходимо найти символ / (слэш). Он используется со знаком \ (обратный слэш), что не будет считаться ошибкой, так как в данном случае / не является концом регулярного выражения. Суффикс {0,3} означает, что символ / может встретиться 0, 1, 2 или 3 раза.

```
([0-9.\-A-Za-Z]+)
```

Следующим фрагментом является захватываемая группа номер 3, которая должна соответствовать названию хоста, состоящему из одной или нескольких цифр, букв, символов . или -. Чтобы не путать символ - с дефисом, задающим диапазон, его экранируют обратным слэшем (\-).

```
(?::(\d+))?
```

Следующий фрагмент находит соответствие для номера порта, указывать который не обязательно. Он представляет собой последовательность из одной или нескольких цифр, которым предшествует символ двоеточие (:). Знаки \d означают цифровой символ. Захватываемая группа 4 будет состоять из одной и более цифр.

```
(:?\/([?#]*))?
```

Есть также и другие необязательные группы. Одна из них начинается с символа /. Класс символов [^?#] начинается с ^, это указывает на то, что данный класс включает в себя все символы, кроме ? и #. Знак * показывает, что данные символы могут встретиться несколько раз или же вообще не иметь соответствий.

Здесь допущена некоторая неточность. Класс, включающий все символы, кроме ? и #, включает также и символы конца строки, управляющие символы и множество других символов, которые вообще не используются в данной строке. В большинстве случаев это даст получить желаемый результат, но есть риск получить лишние результаты. Небрежно составленные регулярные выражения являются основным источником брешей в области безопасности, поскольку допустить неточность при написании регулярного выражения гораздо проще, чем этого не сделать.

```
(:?\([#]*))?
```

Далее идет необязательная группа, начинающаяся с символа ?. Она включает в себя захватываемую группу номер 6, которая может содержать несколько символов (кроме #) или не содержать ни одного символа.

```
(?:#(.*))?
```

Последняя необязательная группа начинается с символа #. Здесь знак . означает соответствие любому символу, исключая символы конца строки.

```
$
```

Символ $ означает конец строки. Он сообщает, что следом за URL-адресом нет никакой дополнительной информации.

Все это — фрагменты регулярного выражения `parse_url`[1].

Конечно, можно создавать и более сложные регулярные выражения, нежели `parse_url`, но нежелательно. Лучше всего, когда регулярные выражения короткие и простые. Только тогда можно быть уверенным, что они работают правильно и в случае необходимости их можно будет успешно модифицировать.

Степень совместимости между процессорами языка JavaScript очень высока. Реализация регулярных выражений является *наименее* переносимой частью языка. Сложные и запутанные регулярные выражения, вероятнее всего, приведут к проблемам переносимости. А вложенные регулярные выражения могут стать источником серьезных проблем производительности. Наилучшей стратегией является простота.

Рассмотрим другой пример: регулярное выражение для сопоставления чисел. Числа могут состоять из целой части с необязательным знаком минус, необязательной дробной части и необязательной экспоненциальной части.

```
var parse_number = /A-?\d+(?:\.\d*)?(?:e[+\-]?\d+)?$/i;

var test = function (num) {
    document.writeln(parse_number.test(num));
};
```

[1] Если попробовать еще раз ввести все вместе, можно легко запутаться:

```
/A(?:([A-Za-z]+):)?(\/{0,3})([0-9.\-A-Za-z]+)(?::(\d+))?(?:\/([A?#]*))?(?:\?([A#]*))?(?:#(.
*))?$/
```

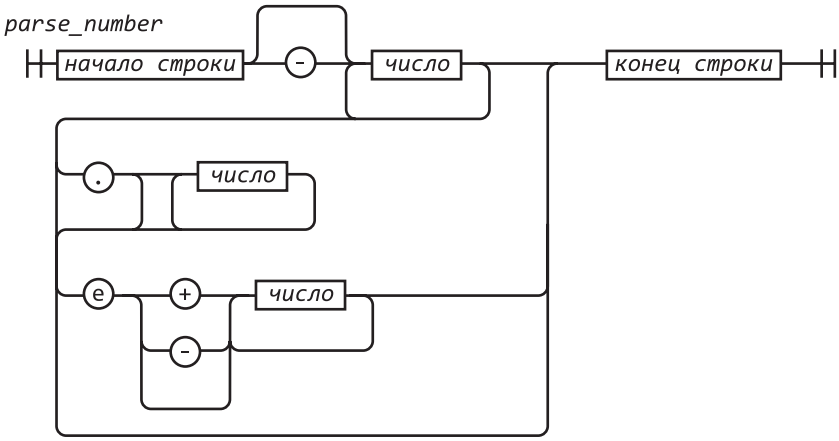
```
test('1');               // верно
test('number');          // не верно
test('98.6');            // верно
test('132.21.86.100');   // не верно
test('123.45E-67');      // верно
test('123.45D-67');      // не верно
```

Выражение `parse_number` успешно идентифицировало строки как соответствующие заданной спецификации, так и все остальные, но не дало никакой информации о том, почему и где оно не смогло установить соответствие.

Разберем выражение `parse_number`.

```
/^ $/i
```

Заключаем регулярное выражение внутрь знаков `^` и `$`. В этом случае все символы в тексте будут сравниваться с регулярным выражением. Если опустить эти якоря, регулярное выражение сообщит, что строка содержит число. С якорями оно сообщит, что строка содержит только число. Если использовать только знак `^`, регулярное выражение будет соответствовать строкам, начинающимся с числа. Если использовать только знак `$`, будут подбираться строки, заканчивающиеся числом.



При использования флага `i` буквы при сравнении игнорируются. Единственная буква в нашем шаблоне — это `e`. Необходимо также, чтобы буква `e` соответствовала `E`. Можно задать фрагмент, обозначающий `e` как `[Ee]` или `(?:E|e)`, но это невозможно из-за наличия флага `i`.

```
-?
```

Суффикс `?` для знака минус указывает, что он не обязателен.

```
\d+
```

Знаки `\d` означают то же, что знаки `[0-9]`, и соответствуют любой цифре. Суффикс `+` означает, что выражение может содержать одну или несколько цифр.

```
(:?\.\d*)?
```

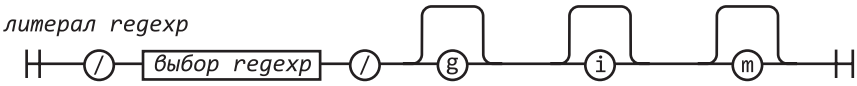
Знаки `(?:...)?` указывают на необязательную не захватываемую группу. Как правило, лучше использовать не захватываемую группу, хотя захватываемая группа и выглядит красивее. Причина в том, что применение захватываемой группы приведет к снижению производительности. Группе будет соответствовать десятичная дробь с несколькими цифрами или вообще без цифр после точки.

```
(:?e[+\-]\d+)?
```

Это еще одна необязательная не захватываемая группа. Ей соответствует знак `e` (или `E`), необязательный знак и одна или несколько цифр.

Конструкция

Есть два пути создания объекта `RegExp`. Как видно из примеров, предпочтительнее использовать литерал регулярного выражения.



Литералы регулярных выражений заключаются между двумя символами слэша, что может вызвать некоторые затруднения, поскольку этот знак используется как оператор деления, а также в комментариях.

Для объекта `RegExp` может быть установлен один из трех флагов, которые обозначаются буквами `g`, `i` и `m`, как показано в табл. 7.1. Флаги добавляются непосредственно в конце литерала `RegExp`:

```
// Создаем объект для регулярного выражения, соответствующего
// JavaScript-строке.

var my_regexp = /"(?:\\.|[A\\\"])*"/g;
```

Таблица 7.1. Флаги регулярных выражений

Флаг	Значение
G	Глобальный (встречается несколько раз, точный смысл меняется в зависимости от метода)
I	Нечувствительный к регистру (регистр букв игнорируется)
M	Многострочный (^ и $ может соответствовать символам конца строки)

Другой способ создать регулярное выражение — использовать конструктор `RegExp`. Конструктор принимает строку и преобразует ее в объект `RegExp`. Следует обратить особое внимание на построение этой строки, так как в регулярных выражениях

обратный слэш имеет несколько иной смысл, чем в строковых литералах. Как правило, необходимо удвоить обратный слэш и использовать кавычки:

```
// Создаем объект для регулярного выражения, соответствующего
// JavaScript-строке.

var my_regexp = new RegExp("'(?:\\\\.|[A\\\\\\\,])*'", 'g'));
```

Вторым параметром является строка, определяющая флаги. Конструктор `RegExp` может пригодиться, когда регулярное выражение должно быть сгенерировано в ходе выполнения программы и использовать недоступные программисту данные.

Объекты `RegExp` содержат свойства, перечисленные в табл. 7.2.

Таблица 7.2. Свойства объекта RegExp

Свойство	Значение свойства
Global	true, если используется флаг g
ignoreCase	true, если используется флаг i
LastIndex	Индекс, с которого при выполнении начинается следующее совпадение. Изначально равен нулю
multiline	true, если используется флаг m
source	Исходный текст регулярного выражения

Объекты `RegExp`, заданные с помощью литералов регулярных выражений, являются частями одного экземпляра объекта:

```
function make_a_matcher() {
    return /a/gi;
}

var x = make_a_matcher();
var y = make_a_matcher();

// Осторожно: x и y — это один и тот же объект!

x.lastIndex = 10;

document.writeln(y.lastIndex);    // 10
```

Элементы

Рассмотрим подробнее элементы, составляющие регулярные выражения.

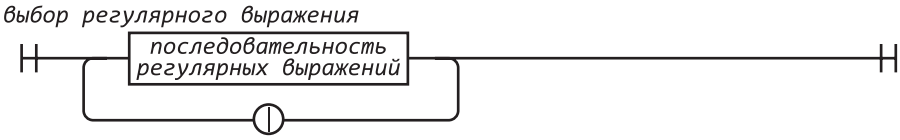
Выбор регулярного выражения

Элемент *выбора регулярного выражения* представляет собой одну или несколько *последовательностей регулярных выражений*. Последовательности разделяются

символом | (вертикальная черта). Соответствие считается найденным, если хотя бы одна из последовательностей имеет соответствие. Соответствие для каждой последовательности ищется по порядку . Итак:

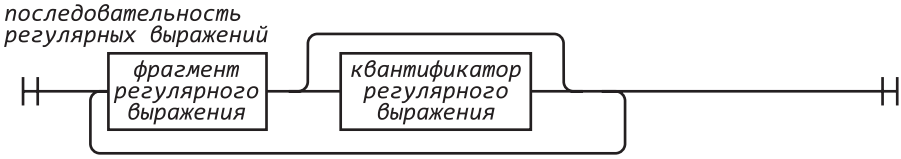
```
"into".match(/in|int/)
```

Найдено соответствие `in` в `into`. Соответствие `int` не ищется, потому что оно уже найдено для `in`.



Последовательность регулярных выражений

Последовательность регулярных выражений состоит из одного или нескольких *фрагментов регулярных выражений*, после каждого из которых дополнительно может содержаться квантификатор, определяющий, сколько раз этот фрагмент должен появляться. Если квантификатор не указан, он считается равным единице.



Фрагмент регулярного выражения

Фрагментом регулярного выражения может быть символ, парные скобки, класс символов или управляющая последовательность. Все символы рассматриваются буквально за исключением управляющих и специальных символов:

```
\ / [] () {} ? + * | . ^ $
```

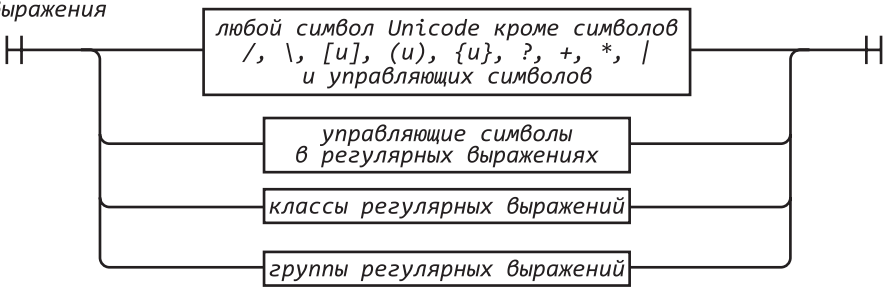
Если следует читать эти символы буквально, то они должны экранироваться префиксом \. Также следует экранировать с помощью префикса \ вызывающие сомнения специальные символы. С буквами или цифрами префикс \ не применяется.

Неэкранированный символ `.` соответствует любому символу кроме символа конца строки.

Неэкранированный символ `^` соответствует началу текста, если значение свойства `lastIndex` равно нулю. Также он может соответствовать символу конца строки, если указан флаг `m`.

Неэкранированный символ `$` соответствует концу текста. Также он может соответствовать символу конца строки, если указан флаг `m`.

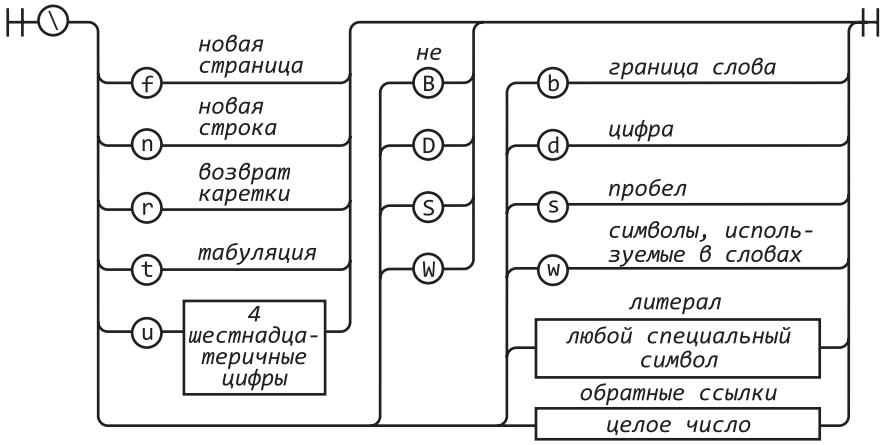
фрагмент регулярного выражения



Управляющие символы регулярных выражений

Обратный слэш служит для управления как фрагментами регулярных выражений, так и строками, но с регулярными выражениями он работает несколько иначе.

управляющие символы в регулярных выражениях



Как и в строках, знаки `\f` означают переход на новую страницу, `\n` — переход на новую строку, `\r` — возврат каретки, `\t` — табуляцию, `\u` — преобразование Unicode-символа в 16-разрядную шестнадцатеричную константу. Для фрагментов регулярных выражений знаки `\b` не удаляют предыдущий символ.

Знаки `\d` так же, как и `[0-9]`, соответствуют цифре. Знаки `\D`, наоборот: `[^0-9]`.

Знаки `\s`, как и `[\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]`, означают неполный набор символов пробелов в Unicode. Знаки `\S`, напротив: `[^\f\n\r\t\u000B\u0020\u00A0\u2028\u2029]`.

Знаки `\w` означают то же, что и `[0-9A-z_a-r]`. Знаки `\W`, наоборот: `[^0-9A-Z_a-z]`. Так должны быть представлены символы, используемые в словах. К сожалению, отвечающий за это класс практически бесполезен для работы с любым реальным

языком. Если требуется задать соответствие буквенному классу, лучше создать собственный класс.

Знаки `[A-Za-z\u00C0-\u1FFF\u2800-\uFFFD]` означают простой буквенный класс, включающий все Unicode-буквы, а также тысячи символов, не являющихся буквами. Стандарт Unicode довольно большой и сложный. Можно задать более точный буквенный класс из символов основной многоязычной матрицы, но он будет сложным и неэффективным. Регулярные выражения в JavaScript очень слабо поддерживают интернационализацию.

Знаки `\b` задумывались как якорь, означающий границы слова и призванный облегчить поиск соответствия по словам. К сожалению, для определения границ слов используются знаки `\w`, что совершенно бесполезно для многоязычных приложений. И это далеко не сильная сторона JavaScript.

Знаки `\1` представляют собой ссылку на текст захватываемой группы номер 1, используемую для повторного поиска соответствий. Например, пусть требуется найти в тексте повторяющиеся слова:

```
var doubled_words =
    /([A-Za-z\u00C0-\u1FFF\u2800-\uFFFD]+)\s+\1/gi;
```

Здесь выражение `doubled_words` ищет вхождения слов (строк, содержащих одну или несколько букв), за которым следует пробел и то же слово.

Знаки `\2` означают ссылку на группу номер 2, знаки `\3` — на группу 3 и т. д.

Группы регулярных выражений

Есть четыре вида групп регулярных выражений:

Захватываемая группа

Захватываемая группа обеспечивает выбор регулярного выражения, заключенного в круглые скобки. При этом символы, соответствующие группе, захватываются. Каждой захватываемой группе присваивается номер. Первая захватываемая группа начинается с символа `(` в регулярном выражении группы номер 1. Вторая начинается с символа `(` в регулярном выражении группы 2.

Не захватываемая группа

Не захватываемая группа имеет префикс `(:`. Не захватываемая группа просто находит соответствия, не захватывая соответствующий текст, что может слегка повысить производительность. Не захватываемые группы не нарушают нумерацию захватываемых групп.

Позитивный просмотр вперед

При позитивном просмотре вперед группа имеет префикс `(?=`. В отличие от не захватываемых групп, такой группе будет найдено соответствие, если для следующей за ней группы также будет найдено соответствие. Это трудно назвать сильной стороной.

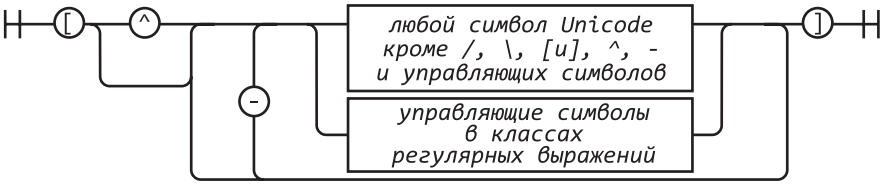
Негативный просмотр вперед

При негативном просмотре вперед группа имеет префикс `(?!`. В отличие от групп с позитивным просмотром вперед, такой группе будет найдено соответствие, если для следующей за ней группы соответствия не будет, что также не является сильной стороной.

Классы регулярных выражений

Класс регулярных выражений — это удобное средство задания набора символов. К примеру, чтобы найти в тексте все гласные, можно было бы написать `(?:a|e|i|o|u)`, но удобнее задать класс `[aeiou]`.

классы регулярных выражений



Классы имеют ряд преимуществ. Первым преимуществом является возможность указать диапазон символов. Например, рассмотрим набор 32 специальных ASCII-символов:

```
! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] A _ ~ { | } ~
```

Этот же набор может быть записан в виде:

```
(?:!|"|#|\$|%|&|'|\(|\)|\*|\+|,|-|\.|\/|:|;|<|=|>|@|\[|\\|]|\^|_|` |\
{|\||\}|~)
```

Однако намного изящнее записать его в виде класса:

```
[!-\/:-@\[-'{-~]
```

Здесь представлены символы от ! до /, от : до @, от [до ', от { до ~. Тем не менее нельзя не отметить, что смотреть на эту мешанину символом по-прежнему не слишком приятно.

Второе преимущество классов: если первый символ после [— это ^, то класс не будет включать указанные символы.

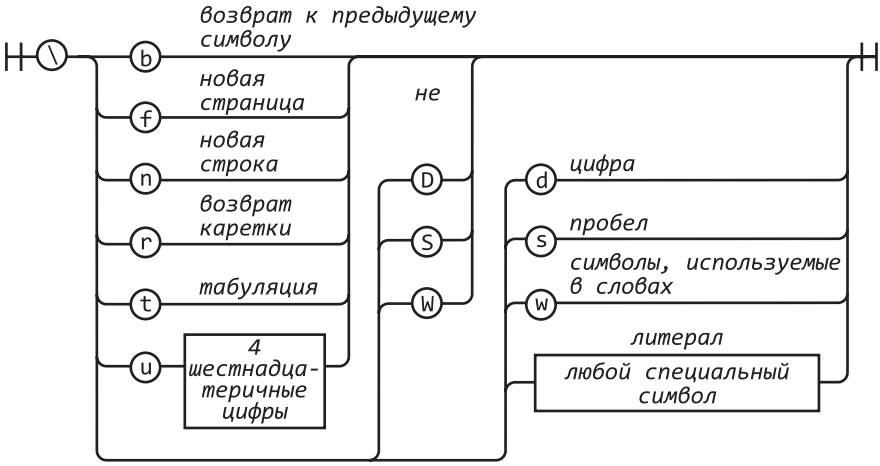
Так, последовательность [^!-\/:-@\[-'{-~] соответствует любому символу, кроме специальных ASCII-символов.

Управляющие символы в классах регулярных выражений

Правила использования управляющих символов в классах немного отличаются от тех, что применяются для фрагментов регулярных выражений. Знаки [\b] означают возврат к предыдущему символу. Вот специальные символы, которые следует экранировать при задании класса символов:

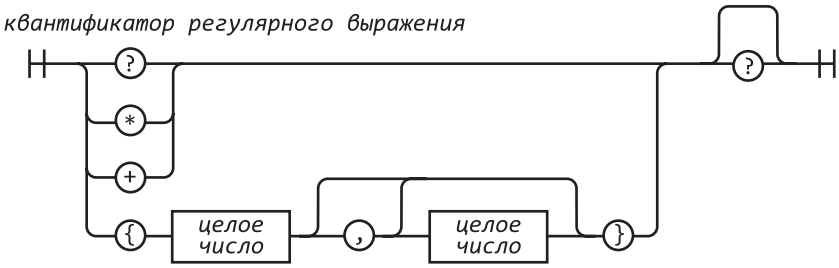
```
- / [ \ ] ^
```

управляющие символы
в классах регулярных выражений



Квантификатор регулярного выражения

Фрагмент регулярного выражения может иметь суффикс, являющийся *квантификатором регулярного выражения* и определяющий, сколько соответствий может иметь данный фрагмент. Число в фигурных скобках означает, что текст, соответствующий фрагменту, может встретиться несколько раз. Таким образом, последовательность /www/ будет соответствовать /w{3}/, последовательность {3,6} — 3, 4, 5 или 6 повторениям, а последовательность {3,} — 3 и более повторениям.



Знак `?` означает то же самое, что `{0,1}`, знак `*` — то же, что `{0}`, а знак `+` — то же, что `{1}`.

Квантификация может быть жадной, в этом случае поиск соответствий будет продолжаться до последнего повторения. Если у квантификатора есть дополнительный суффикс `?`, то квантификация является ленивой, в этом случае поиск соответствий будет производиться до минимально возможного числа повторений. Обычно лучше задействовать жадную квантификацию.

8 Методы

Хотя это и безумие, метод все же есть.

Уильям Шекспир. Гамлет

JavaScript обладает небольшим набором готовых методов, доступных для стандартных типов данных.

Массивы

array.concat(item...)

Метод `concat` создает новый массив, содержащий копию массива `array`, дополненную элементами `item`. Если `item` является массивом, то добавляются только те элементы, которых еще нет в исходном массиве. Далее в этой главе описан еще один похожий метод `array.push(item...)`.

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.concat(b, true);
// массив c равен ['a', 'b', 'c', 'x', 'y', 'z', true]
```

array.join(separator)

Метод `join` преобразует `array` в строку. Каждый элемент массива преобразуется в строку, затем строки объединяются вместе через разделитель `separator`. По умолчанию `separator` имеет значение `','`. Для того чтобы объединить строки без разделителя, необходимо задать в качестве значения `separator` пустую строку.

Если нужно получить строку из множества частей, может быть быстрее собрать их в массив и использовать метод join, чем объединять все части с помощью оператора +:

```
var a = ['a', 'b', 'c'];
a.push('d');
var c = a.join('');    // строка c равна 'abcd';
```

array.pop()

Методы pop и push позволяют работать с массивом как со стеком. Метод pop удаляет из array последний элемент и возвращает его в качестве результата. Если массив array пуст, то возвращается значение undefined.

```
var a = ['a', 'b', 'c'];
var c = a.pop();    // массив a равен ['a', 'b'], c равно 'c'
```

Метод pop может быть реализован следующим образом:

```
Array.method('pop', function () {
    return this.splice(this.length - 1, 1)[0];
});
```

array.push(item...)

Метод push добавляет элемент item в конец массива. В отличие от метода concat, он модифицирует исходный массив array и добавляет сразу все элементы массива. Метод push возвращает новое значение длины массива array:

```
var a = ['a', 'b', 'c'];
var b = ['x', 'y', 'z'];
var c = a.push(b, true);
// массив a равен ['a', 'b', 'c', ['x', 'y', 'z'], true]
// c равно 5;
```

Метод push может быть реализован следующим образом:

```
Array.method('push', function () {
    this.splice.apply(
        this,
        [this.length,0].concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

array.reverse()

Метод reverse переставляет элементы массива array в обратном порядке и возвращает новое значение array:

```
var a = ['a', 'b', 'c'];
var b = a.reverse();
// оба массива a и b равны ['c', 'b', 'a']
```

array.shift()

Метод `shift` удаляет первый элемент `array` и возвращает его значение. Если массив пуст, то возвращается значение `undefined`. Метод `shift`, как правило, гораздо медленнее, чем `pop`:

```
var a = ['a', 'b', 'c'];
var c = a.shift();    // массив a равен ['b', 'c'], c равно 'a'
```

Метод `shift` может быть реализован следующим образом:

```
Array.method('shift', function () {
    return this.splice(0, 1)[0];
});
```

array.slice(start, end)

Метод `slice` создает копию части `array`, начиная с элемента `array[start]` и заканчивая элементом `array[end]`. Задавать параметр `end` не обязательно, по умолчанию он равен `array.length`. Если один из параметров отрицателен, чтобы получить неотрицательное значение, к нему будет добавлено значение `array.length`. Если параметр `start` больше или равен `array.length`, метод возвращает новый пустой массив. Не путайте методы `slice` и `splice`. Метод `string.slice` описан далее в этой главе.

```
Var a = ['a', 'b', 'c'];
Var b = a.slice(0, 1);    // массив b равен ['a']
Var c = a.slice(1);       // массив c равен ['b', 'c']
Var d = a.slice(1, 2);    // массив d равен ['b']
```

array.sort(comparefn)

Метод `sort` упорядочивает элементы массива `array`, но числа он сортирует неправильно:

```
var n = [4, 8, 15, 16, 23, 42];
n.sort();
// массив n равен [15, 16, 23, 4, 42, 8]
```

Стандартная функция сравнения в JavaScript предполагает, что элементы, которые требуется отсортировать, являются строками. Эта функция не так хороша, чтобы перед сравнением проверять тип элементов, она преобразует числа в строки и сравнивает их, предоставляя на удивление неправильные результаты.

К счастью, можно заменить функцию сравнения собственной, которая будет принимать два параметра и возвращать 0, если они равны; отрицательное число, если сначала должен идти первый параметр; и положительное число, если сначала должен быть второй. (Некоторым старожилам это может напомнить арифметическую инструкцию `IF` в FORTRAN II.)

```
n.sort(function (a, b) {
    return a - b;
});
// массив n равен [4, 8, 15, 16, 23, 42];
```

Эта функция служит для сортировки чисел, но не строк. Если есть необходимость отсортировать массив, элементы которого могут принадлежать к любому простому типу, придется немного потрудиться:

```
var m = ['aa', 'bb', 'a', 4, 8, 15, 16, 23, 42];
m.sort(function (a, b) {
    if (a === b) {
        return 0;
    }
    if (typeof a === typeof b) {
        return a < b ? -1 : 1;
    }
    return typeof a < typeof b ? -1 : 1;
});
// массив m имеет вид [4, 8, 15, 16, 23, 42, 'a', 'aa', 'bb']
```

Если не оговорено специально, то перед сравнением функция должна приводить операнды к нижнему регистру. Далее в этой главе описан похожий метод `string.localeCompare`.

Более совершенная функция сравнения может позволить сортировать массив объектов. Для того чтобы упростить задачу, напишем функцию, создающую функции сравнения:

```
// Функция by принимает строку с именем члена объекта и возвращает
// функцию сравнения, которая может быть использована для сортировки
// массива объектов, содержащих этот член.

var by = function (name) {
    return function (o, p) {
        var a, b;
        if (typeof o === 'object' && typeof p === 'object' && o && p) {
            a = o[name];
            b = p[name];
            if (a === b) {
                return 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
            return typeof a < typeof b ? -1 : 1;
        } else {
            throw {
                name: 'Error',
                message: 'Expected an object when sorting by ' + name;
            };
        }
    };
};
```

продолжение ⇩

```
var s = [
    {first: 'Joe',   last: 'Besser'},
    {first: 'Moe',   last: 'Howard'},
    {first: 'Joe',   last: 'DeRita'},
    {first: 'Shemp', last: 'Howard'},
    {first: 'Larry', last: 'Fine'},
    {first: 'Curly', last: 'Howard'}
];

s.sort(by('first'));    // массив s равен [
//    {first: 'Curly',    last: 'Howard'},
//    {first: 'Joe',      last: 'DeRita'},
//    {first: 'Joe',      last: 'Besser'},
//    {first: 'Larry',    last: 'Fine'},
//    {first: 'Moe',      last: 'Howard'},
//    {first: 'Shemp',    last: 'Howard'}
// ]
```

Метод `sort` недостаточно стабилен, так следующая инструкция не гарантирует правильной последовательности:

```
s.sort(by('first')).sort(by('last'));
```

Если необходима сортировка по нескольким ключам, придется еще немного потрудиться. Можно изменить функцию `by` и передать ей два параметра, создать другой метод сравнения, вызываемый для разрыва связей после того, как произошло сравнение с главным ключом:

```
// Функция by принимает строку с именем члена объекта
// и необязательную функцию сравнения minor и возвращает
// функцию сравнения, которая может быть использована для сортировки
// массива объектов, содержащих этот член.
// Функция сравнения minor используется для разрыва связей,
// если o[name] и p[name] равны.

var by = function (name, minor) {
    return function (o, p) {
        var a, b;
        if (o && p && typeof o === 'object' && typeof p === 'object') {
            a = o[name];
            b = p[name];
            if (a === b) {
                return typeof minor === 'function' ? minor(o, p) : 0;
            }
            if (typeof a === typeof b) {
                return a < b ? -1 : 1;
            }
            return typeof a < typeof b ? -1 : 1;
        } else {
```

```
            throw {
                name: 'Error',
                message: 'Expected an object when sorting by ' + name;
            };
        };
    };
};

s.sort(by('last', by('first')));        // массив s равен [
//    {first: 'Joe',      last: 'Besser'},
//    {first: 'Joe',      last: 'DeRita'},
//    {first: 'Larry',    last: 'Fine'},
//    {first: 'Curly',    last: 'Howard'},
//    {first: 'Moe',      last: 'Howard'},
//    {first: 'Shemp',    last: 'Howard'}
// ]
```

array.splice(start, deleteCount, item...)

Метод `splice` удаляет элементы массива, заменяя их новыми элементами `item`. Параметр `start` — это индекс элемента в массиве `array`, с которого следует начать удаление. Параметр `deleteCount` — это количество удаляемых элементов, начиная с указанной позиции. Если заданы дополнительные параметры, элементы `item` вставляются в указанную позицию. Метод возвращает массив, содержащий удаленные элементы.

Чаще всего `splice` используется для удаления элементов из массива. Не путайте методы `splice` и `slice`:

```
var a = ['a', 'b', 'c'];
var r = a.splice(1, 1, 'ache', 'bug');
// массив a равен ['a', 'ache', 'bug', 'c']
// массив r равен ['b']
```

Метод `splice` может быть реализован так:

```
Array.method('splice', function (start, deleteCount) {
    Var max = Math.max,
        min = Math.min,
        delta,
        element,
        insertCount = max(arguments.length - 2, 0),
        k = 0,
        len = this.length,
        new_len,
        result = [],
        shift_count;
    start = start || 0;
    if (start < 0) {
```

продолжение ⇩

```
        start += len;
    }
    start = max(min(start, len), 0);
    deleteCount = max(min(typeof deleteCount === 'number' ?
            deleteCount : len, len - start), 0);
    delta = insertCount - deleteCount;
    new_len = len + delta;
    while (k < deleteCount) {
        element = this[start + k];
        if (element !== undefined) {
            result[k] = element;
        }
        k += 1;
    }
    shift_count = len - start - deleteCount;
    if (delta < 0) {
        k = start + insertCount;
        while (shift_count) {
            this[k] = this[k - delta];
            k += 1;
            shift_count -= 1;
        }
        this.length = new_len;
    } else if (delta > 0) {
        k = 1;
        while (shift_count) {
            this[new_len - k] = this[len - k];
            k += 1;
            shift_count -= 1;
        }
        this.length = new_len;
    }
    for (k = 0; k < insertCount; k += 1) {
        this[start + k] = arguments[k + 2];
    }
    return result;
});
```

array.unshift(item...)

Метод `unshift`, в отличие от `push`, добавляет элемент `item` в начало массива `array`, а не в конец. Метод возвращает новое значение `length` массива `array`:

```
var a = ['a', 'b', 'c'];
var r = a.unshift('?', '@');
// массив a равен ['?',    'a', 'b', 'c']
// массив r равен 5
```

Метод unshift может быть реализован так:

```
Array.method('unshift', function ( ) {
    this.splice.apply(this,
        [0, 0].concat(Array.prototype.slice.apply(arguments)));
    return this.length;
});
```

Функции

function.apply(thisArg, argArray)

Метод apply вызывает функцию, в которую передает объект, связанный с this, и необязательный массив аргументов. Метод apply используется в программном шаблоне вызова метода apply (см. главу 4):

```
Function.method('bind', function (that) {

// Возвращаем функцию, вызывающую эту функцию как метод объекта that.

    var method = this,
        slice = Array.prototype.slice,
        args = slice.apply(arguments, [1]);
    return function ( ) {
        return method.apply(that,
            args.concat(slice.apply(arguments, [0])));
        };
    });

    var x = function () {
        return this.value;
    }.bind({value: 666});
    alert(x());     // 666
```

Числа

number.toExponential(fractionDigits)

Метод toExponential преобразует число number в строку в экспоненциальной форме. Необязательный параметр fractionDigits задает количество знаков после запятой в диапазоне от 0 до 20:

```
document.writeln(Math.PI.toExponential(o));
document.writeln(Math.PI.toExponential(2));
document.writeln(Math.PI.toExponential(7));
document.writeln(Math.PI.toExponential(16));
document.writeln(Math.PI.toExponential());
```

продолжение ⇩

```
// Получаем

3e+0
3.14e+0
3.1415927e+0
3.1415926535897930e+0
3.141592653589793e+0
```

number.toFixed(fractionDigits)

Метод `toFixed` преобразует число `number` в строку в десятичной форме. Необязательный параметр `fractionDigits` задает количество знаков после запятой в диапазоне от 0 до 20. По умолчанию он равен 0:

```
document.writeln(Math.PI.toFixed(0));
document.writeln(Math.PI.toFixed(2));
document.writeln(Math.PI.toFixed(7));
document.writeln(Math.PI.toFixed(16));
document.writeln(Math.PI.toFixed( ));

// Получаем

3
3.14
3.1415927
3.1415926535897930
3
```

number.toPrecision(precision)

Метод `toPrecision` преобразует число `number` в строку в десятичной форме. Необязательный параметр `precision` задает число значащих цифр в диапазоне от 1 до 21:

```
document.writeln(Math.PI.toPrecision(2));
document.writeln(Math.PI.toPrecision(7));
document.writeln(Math.PI.toPrecision(16));
document.writeln(Math.PI.toPrecision( ));

// Получаем

3.1
3.141593
3.141592653589793
3.141592653589793
```

number.toString(radix)

Метод `toString` преобразует число `number` в строку. Необязательный параметр `radix` задает основание системы счисления в диапазоне от 2 до 36. По умолчанию

значение `radix` равно 10. Параметр `radix` чаще используется с целыми числами, но не обязательно.

В большинстве случаев `number.toString()` можно записать проще, как `String(number)`:

```
document.writeln(Math.PI.toString(2));
document.writeln(Math.PI.toString(8));
document.writeln(Math.PI.toString(16));
document.writeln(Math.PI.toString());

// Получаем

11.001001000011111101101010100010001000010110100011
3.1103755242102643
3.243f6a8885a3
3.141592653589793
```

Объекты

object.hasOwnProperty(name)

Метод `hasOwnProperty` возвращает `true`, если объект обладает свойством `name`. Цепочка прототипов в данном методе не рассматривается, к тому же он бесполезен, если параметр `name` — это `hasOwnProperty`:

```
var a = {member: true};
var b = Object.create(a);               // см. главу 3
var t = a.hasOwnProperty('member');     // t имеет значение true
var u = b.hasOwnProperty('member');     // u имеет значение false
var v = b.member;                       // v имеет значение true
```

Регулярные выражения

regexp.exec(sting)

Метод `exec` является самым мощным (и самым медленным) методом для работы с регулярными выражениями. Если выражение `regexp` соответствует строке `string`, то метод возвращает массив. Нулевой элемент массива будет содержать подстроку, соответствующую регулярному выражению, первый элемент — текст захватываемой группы номер 1, второй элемент — текст захватываемой группы 2 и т. д. Если найти соответствие не удается, метод возвращает `null`.

Если выражение `regexp` имеет флаг `g`, ситуация немного усложняется. Поиск начинается не с нулевой позиции в строке, а с позиции `regexp.lastIndex`

(которая считается нулевой). Если соответствие успешно найдено, то индекс `regexp.lastIndex` устанавливается в позицию символа, следующего за фрагментом, где было обнаружено соответствие. В случае неудачи значение `regexp.lastIndex` сбрасывается в 0.

Вызов `exec` в цикле позволяет найти в строке несколько вхождений шаблона. Важно не упустить следующее: чтобы досрочно выйти из цикла, перед очередным входом в цикл необходимо самостоятельно сбросить значение `regexp.lastIndex` в 0. Кроме того, начать поиск соответствий с начала можно только при нулевом значении `regexp.lastIndex`:

```
// Разобьем простой HTML-текст на теги и текст.
// (См. string.replace для метода entityify.)
// Для каждого тега или текста создается массив, содержащий:
// [0] полное соответствие тегу или тексту
// [1] символ /, если он есть
// [2] имя тега
// [3] атрибуты, если таковые имеются

var text = '<html><body bgcolor=linen><p>' +
        'This is <b>bold<\/b>!<\/p><\/body><\/html>';

var tags = /[A<>]+|<(\/?)([A-Za-z]+)([A<>]*)>/g;
var a, i;

while ((a = tags.exec(text))) {
    for (i = 0; i < a.length; i += 1) {
        document.writeln(('// [' + i + '] ' + a[i]).entityify());
    }
    document.writeln( );
}

// Результат:

// [0] <html>
// [1]
// [2] html
// [3]

// [0] <body bgcolor=linen>
// [1]
// [2] body
// [3] bgcolor=linen

// [0] <p>
// [1]
// [2] p
// [3]
```

```
// [0] This is
// [1] undefined
// [2] undefined
// [3] undefined

// [0] <b>
// [1]
// [2] b
// [3]

// [0] bold
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </b>
// [1] /
// [2] b
// [3]

// [0] !
// [1] undefined
// [2] undefined
// [3] undefined

// [0] </p>
// [1] /
// [2] p
// [3]

// [0] </body>
// [1] /
// [2] body
// [3]

// [0] </html>
// [1] /
// [2] html
// [3]
```

regexp.test(string)

Метод `test` — самый простой (и самый быстрый) метод для работы с регулярными выражениями. Если выражение `regexp` соответствует строке `string`, он возвращает `true`, в противном случае — `false`. Не используйте с этим методом флаг `g`:

```
var b = /&.+;/.test('frank &amp; beans');
// b имеет значение true
```

Метод `test` может быть реализован так:

```
RegExp.method('test', function (string) {
    return this.exec(string) !== null;
});
```

Строки

string.charAt(pos)

Метод `charAt` возвращает символ из строки `string` в позиции `pos`. Если значение `pos` меньше нуля или больше либо равно `string.length`, вернет пустую строку. В JavaScript нет символьного типа. Результатом вызова этого метода является строка:

```
var name = 'Curly';
var initial = name.charAt(0);    // initial равна 'C'
```

Метод `charAt` может быть реализован так:

```
String.method('charAt', function ( pos) {
    return this.slice(pos, pos + 1);
});
```

string.charCodeAt(pos)

Метод `charCodeAt`, в отличие от `charAt`, вместо строки возвращает целое значение кода символа из строки `string` в позиции `pos`. Если значение `pos` меньше нуля или больше либо равно `string.length`, метод возвращает `NaN`:

```
var name = 'Curly';
var initial = name.charCodeAt(0);    // initial имеет значение 67
```

string.concat(string...)

Метод `concat` создает новую строку путем объединения других строк. Поскольку оператор `+` гораздо удобнее, метод `concat` используется редко:

```
var s = 'C'.concat('a', 't');     // s равно 'Cat'
```

string.indexOf(searchString, position)

Метод `indexOf` ищет подстроку `SearchString` в строке `string`. Если она находится, то метод возвращает позицию первого символа подстроки, в противном случае он возвращает –1. Необязательный параметр `position` задает позицию, с которой следует начать поиск в строке `string`:

```
var text = 'Mississippi';
var p = text.indexOf('ss');   // p имеет значение 2
p = text.indexOf('ss', 3);    // p имеет значение 5
p = text.indexOf('ss', 6);    // p имеет значение -1
```

string.lastIndexOf(searchString, position)

Метод `lastIndexOf`, в отличие от `IndexOf`, начинает поиск с конца строки, а не с начала:

```
var text = 'Mississippi';
var p = text.lastIndexOf('ss');   // p имеет значение 5
p = text.lastIndexOf('ss', 3);    // p имеет значение 2
p = text.lastIndexOf('ss', 6);    // p имеет значение 5
```

string.localeCompare(that)

Метод `localeCompare` сравнивает две строки. Правила сравнения строк не указываются. Если строка `that` меньше `string`, результат отрицательный. Если строки равны, то результат равен нулю. Этот метод напоминает функцию сравнения `array.sort`:

```
var m = ['AAA', 'A', 'aa', 'a', 'Aa', 'aaa'];
m.sort(function (a, b) {
    return a.localeCompare(b);
});
//     m (в некотором случае) равен
//         ['a', 'A', 'aa', 'Aa', 'aaa', 'AAA']
```

string.match(regexp)

Метод `match` ищет соответствие между строкой и регулярным выражением. Как он это делает, зависит от флага `g`. Если флаг `g` не указан, то результат вызова `string.match(regexp)` такой же, как при вызове `regexp.exec(string)`. Однако если регулярное выражение имеет флаг `g`, то метод `match` возвращает массив со всеми соответствиями за исключением захватываемых групп:

```
var text = '<html><body bgcolor=linen><p>' +
    'This is <b>bold<\/b>!<\/p><\/body><\/html>';
var tags = /[A<>]+|<(\/?)([A-Za-z]+)([A<>]*)>/g;
var a, i;

a = text.match(tags);
for (i = 0; i < a.length; i += 1) {
    document.writeln(('// [' + i + '] ' + a[i]).entityify());
}
```

продолжение ⇩

```
// Результат:

// [0] <html>
// [1] <body bgcolor=linen>
// [2] <p>
// [3] This is
// [4] <b>
// [5] bold
// [6] </b>
// [7] !
// [8] </p>
// [9] </body>
// [10] </html>
```

string.replace(searchValue, replaceValue)

Метод `replace` производит операции поиска и замены внутри строки `string`, создавая новую строку. Аргумент `searchValue` может быть строкой или объектом регулярного выражения. Если это строка, то заменяется только первое вхождение значения `searchValue`:

```
var result = "mother_in_law".replace('_', '-');
```

В результате мы, к несчастью, получаем строку `"mother-in_law"`.

Если аргумент `searchValue` является регулярным выражением и имеет флаг `g`, то заменяются все вхождения. Если флага `g` нет, то метод заменяет только первое вхождение.

Аргумент `replaceValue` может быть строкой или функцией. Если `replaceValue` — строка, символ `$` имеет особое значение (табл. 8.1):

```
// 3 цифры, заключенные в скобки

var oldareacode = /\((\d{3})\)/g;
var p = '(555)666-1212'.replace(oldareacode, '$1-');
// p равно '555-666-1212'
```

Таблица 8.1. Значения символа $

Последовательность со знаком $	Замена
$$	$
$&	Соответствующий текст
$number	Текст захватываемой группы
$`	Текст, предшествующий соответствию
$'	Текст, следующий за соответствием

Если аргумент `replaceValue` является функцией, то она вызывается для каждого соответствия, и строка, возвращаемая функцией, используется в качестве текста замены. Первый параметр передает функции соответствующий текст. Вторым

параметром является текст захватываемой группы 1, следующий параметр — это текст захватываемой группы 2 и т. д.:

```
String.method('entityify', function ( ) {

var character = {
    '<' : '&lt;',
    '>' : '&gt;',
    '&' : '&amp;',
    '"' : '&quot;'
};

// Возвращаем метод string.entityify, который, в свою очередь,
// возвращает результат вызова метода replace.
// Функция replaceValue возвращает результат поиска символа в объекте.
// Это бывает необходимо при использовании инструкции switch.

    return function ( ) {
        return this.replace(/[<>&"]/g, function (c) {
            return character[c];
        });
    };
}());
alert("<&>".entityify());    // &lt;&amp;&gt;
```

string.search(regexp)

Метод `search`, в отличие от `indexOf`, получает в качестве параметра объект регулярного выражения, а не строку. Он возвращает позицию первого символа первого соответствия, если таковое имеется, или –1, если поиск не удался. Флаг `g` игнорируется. Метод не имеет параметра `position`:

```
var text = 'and in it he says "Any damn fool could';
var pos = text.search(/["']/);    // pos имеет значение 18
```

string.slice(start, end)

Метод `slice` создает новую строку, копируя часть строки `string`. Если значение параметра `start` отрицательно, к нему добавляется величина `string.length`. Параметр `end` является необязательным, по умолчанию его значение равно `string.length`. Если значение параметра `end` отрицательно, к нему добавляется величина `string.length`. Значение `end` на единицу больше позиции последнего символа. Для того чтобы получить `n` символов, начиная с позиции `p`, следует использовать метод `string.slice(p, p+n)`. В конце и начале этой главы описаны похожие методы `string.substring` и `array.slice`.

```
var text = 'and in it he says "Any damn fool could';
var a = text.slice(18);
```

продолжение ⇩

```
// a равна '"Any damn fool could'
var b = text.slice(0, 3);
// b равна 'and'
var c = text.slice(-5);
// c равна 'could'
var d = text.slice(19, 32);
// d равна 'Any damn fool'
```

string.split(separator, limit)

Метод `split` создает массив строк, разделяя на части строку `string`. Необязательный параметр `limit` может ограничить количество этих частей. Параметр `separator` может быть строкой или регулярным выражением.

Если разделитель `separator` — это пустая строка, получаем массив символов:

```
var digits = '0123456789';
var a = digits.split('', 5);
// массив a равен ['0', '1', '2', '3', '4']
```

В противном случае в строке `string` ищутся все вхождения разделителя `separator`. Каждая единица текста между разделителями копируется в массив. Флаг `g` игнорируется:

```
var ip = '192.168.1.0';
var b = ip.split('.');
// массив b равен ['192', '168', '1', '0']

var c = '|a|b|c|'.split('|');
// массив c равен ['', 'a', 'b', 'c', '']

var text = 'last, first ,middle';
var d = text.split(/\s*,\s*/);
// массив d равен [
//     'last',
//     'first',
//     'middle'
// ]
```

Обратите внимание на следующий случай: текст захватываемой группы включается в результат:

```
var e = text.split(/\s*(,)\s*/);
// массив e равен [
//     'last',
//     ',',
//     'first',
//     ',',
//     'middle'
// ]
```

Существуют реализации, исключающие пустые строки из получаемого массива, если разделитель `separator` является регулярным выражением.

```
var f = '|a|b|c|'.split(/\|/);
// в некоторых системах массив f равен ['a', 'b', 'c'],
// в других массив f равен ['', 'a', 'b', 'c', ''].
```

string.substring(start, end)

Метод `substring`, в отличие от `slice`, не обрабатывает отрицательные параметры. Поэтому лучше использовать метод `slice`, а не `substring`.

string.toLocaleLowerCase()

Метод `toLocaleLowerCase` создает новую строку, которая получается путем приведения строки `string` к нижнему регистру в соответствии с правилами локализации. Это может быть полезно для турецкого языка, где `'I'` превращается в 1, а не `'i'`.

string.toLocaleUpperCase()

Метод `toLocaleUpperCase` создает новую строку, которая получается путем приведения строки `string` к верхнему регистру в соответствии с правилами локализации. Это может быть полезно для турецкого языка, где `'i'` превращается в `'İ'`, а не в `'I'`.

string.toLowerCase()

Метод `toLowerCase` создает новую строку, которая получается путем приведения строки `string` к нижнему регистру.

string.toUpperCase()

Метод `toLocaleUpperCase` создает новую строку, которая получается путем приведения строки `string` к верхнему регистру.

String.fromCharCode(char...)

Функция `String.fromCharCode` создает строку из последовательности чисел.

```
var a = String.fromCharCode(67, 97, 116);
// a равно 'Cat'
```

9 Стиль

Дурацкая напыщенная речь!

Уильям Шекспир. Генрих VI. Часть 1

Программирование — далеко не самая простая вещь, придуманная человеком. Программы состоят из огромного количества частей, представленных в определенной последовательности в виде функций, инструкций и выражений, так чтобы теоретически не содержать ошибок. Процесс выполнения имеет мало общего с реализующей его программой. Как правило, ожидается, что программное обеспечение на протяжении своего жизненного цикла будет обновляться. Однако преобразование одной корректной программы в другую, не менее корректную, — задача чрезвычайно сложная.

Хорошие программы не слишком загромождены, они имеют ясную структуру, предвосхищающую возможные изменения. Четко выстроенная программа повышает шансы понимать код, успешно его модифицировать или исправлять ошибки.

Все эти замечания справедливы для любых языков программирования, и особенно для JavaScript. Для того чтобы компенсировать нестрогую типизацию и слишком поверхностное отношение к ошибкам, вызванное недостатком времени на компиляцию, а также гарантировать качество программ, следует придерживаться строгой организации кода.

В JavaScript довольно много недостатков, способных свести на «нет» любые попытки создания хороших программ. Очевидно, что недостатки нужно пытаться обойти. Иногда рискованно использовать даже те средства, которые могут быть полезны, но ненадежны. Такие средства обладают свойством притягивать проблемы, и отказываясь от них, можно исключить довольно широкий класс потенциальных ошибок.

Продолжительность использования какого-либо программного обеспечения напрямую зависит от качества кода. За свой жизненный цикл программа проходит не через одну пару рук и глаз. Когда структура и характеристики представлены

предельно ясно, в случае если в далеком будущем потребуется внести в нее какие-либо изменения, вероятность краха программы будет ниже.

Зачастую JavaScript-код предназначен непосредственно для публикации, что требует от него определенного уровня качества и аккуратности изложения. Четко и последовательно написанные программы удобнее читать.

Программисты могут бесконечно спорить о том, что такое хороший стиль. Большинство программистов зациклились на том, к чему привыкли, поэтому у них преобладает стиль, который они использовали в школе или на своей первой работе. Некоторые из них построили свою карьеру без малейшего чувства стиля. Но разве это не доказывает, что стиль не имеет никакого значения? А если стиль не так важен, может ли один стиль быть лучше, чем любой другой?

Оказывается, что стиль в программировании имеет то же значение, что и в письме. Он нужен для удобства чтения.

Иногда компьютерные программы разрабатываются для внутренней среды, поэтому, пока программа работает, совершенно не важно, как она написана. Тем не менее оказывается, что вероятность того, что понятно написанная программа будет работать, выше, а это, в свою очередь, повышает вероятность того, что программа будет работать именно так, как предполагалось. Кроме того, в течение своего жизненного цикла программное обеспечение может претерпевать колоссальные изменения. И если программисты смогут успешно читать и понимать код, можно надеяться, что они смогут его изменить и улучшить.

В этой книге при написании примеров я использовал единый стиль. Моей целью было создать код, который, по возможности, будет легко читать. Для того чтобы акцентировать внимание на смысле программ, я не жалел дополнительных пробелов.

Содержимое блоков и литералов объектов я выделял отступом в четыре пробела. Я оставлял пространство между символами `if` и `(`, чтобы инструкция `if` не была похожа на вызов функции. Только в вызовах символ `(` идет сразу за предыдущим символом. Я оставлял пробелы вокруг всех инфиксных операторов, исключая операторы `.` и `[`, которым из-за более высокого приоритета это не требуется. Я вставлял пробелы после каждой запятой и двоеточия.

Я оставлял не более чем по одной инструкции в строке. Несколько инструкций в строке могут вызвать ошибку. Если инструкция не помещалась в строке, после запятой или бинарного оператора я переносил ее на следующую. Это позволяет защититься от ошибок копирования и вставки, возникающих из-за механизма автоматического дополнения строк точками с запятой. (Вся трагичность этой ситуации описана в приложении А.) После переноса части инструкции я делал отступ из четырех пробелов или восьми, если наличие четырех пробелов добавляло неоднозначности (например, при переносе строки с инструкцией `if`).

Чтобы избежать ошибок, я всегда использовал блоки структурированных инструкций, таких как `if` и `while`. Например:

```
if (a)
    b();
```

Я видел, что этот текст может стать таким:

```
if (a)
    b();
    c();
```

То есть превратиться в ошибку, которую будет очень трудно обнаружить. Это похоже на следующий фрагмент:

```
if (a) {
    b();
    c();
}
```

Однако означает он совсем другое:

```
if (a) {
    b();
}
c();
```

Код, который выглядит как одно, а на самом деле означает другое, является источником ошибок. Фигурные скобки — довольно простое средство защиты от ошибок, обнаружить которые бывает не так уж просто.

Я всегда использую K&R-стиль и ставлю символ `{` в конце строки, а не в начале, потому что это позволяет избежать ужасной ошибки, связанной с применением JavaScript-инструкции `return`.

Я включил в код комментарии. Мне нравится помещать в своих программах комментарии и оставлять в них информацию, которую потом смогут прочесть другие (а возможно и я сам), чтобы понять, о чем я думал. Иногда комментарии мне кажутся своеобразной машиной времени, которую я использую, чтобы слать себе в будущее важные сообщения.

Я изо всех сил старался сделать комментарии актуальными. Ошибки в комментариях могут еще сильнее затруднить чтение и понимание программы, а этого я себе позволить не могу.

Я старался не тратить время на бесполезные комментарии вроде этого:

```
i = 0; // Устанавливает для i значение ноль.
```

В JavaScript я предпочитаю использовать однострочные комментарии. Блочные комментарии я оставляю для формального документирования и комментирования вывода.

Я предпочитаю создавать такую программную структуру, которая говорила бы сама за себя, и комментарии были бы излишне. Но это не всегда удается, поэтому, пока мои программы не достигли совершенства, я продолжу писать комментарии.

Язык JavaScript обладает синтаксисом C, но у его блоков не существует области видимости. Таким образом, соглашение о том, что переменные должны объявляться

при их первом использовании, в JavaScript неприменимо. В JavaScript предусмотрена область видимости для функции, но не для блоков, поэтому я объявляю все необходимые переменные в начале каждой функции. JavaScript позволяет объявлять переменные уже после их использования. Для меня это выглядит как ошибка, а я не хочу создавать код, даже просто похожий на неправильный. Я хочу, чтобы всегда можно было заметить ошибку. Также я никогда не использую выражение присваивания в условной части инструкции `if`:

```
if (a = b) { ... }
```

Здесь вероятно предполагается следующее:

```
if (a === b) { ... }
```

А я хочу избежать идиом, похожих на ошибки.

Используя инструкцию `switch`, я всегда исключаю переход к выполнению следующего блока `case`. Однажды сразу после пламенной речи о том, почему иногда полезно бывает переход к следующему блоку `case`, я нашел в своем коде ошибку, вызванную таким непреднамеренным переходом. Мне повезло, что я умею учиться на собственном опыте. Сегодня при описании особенностей языка я стараюсь обратить особое внимание на возможности, которые иногда могут быть довольно полезны, но иногда задействовать их бывает просто опасно. И это *хуже всего*, ведь так трудно сказать, используются ли они правильно или в них кроются ошибки.

Вероятно, при разработке, реализации и стандартизации JavaScript высокое качество во главу угла не ставилось, что, несомненно, увеличивает нагрузку на пользователей, вынужденных бороться с недостатками этого языка.

Язык JavaScript поддерживает создание больших программ, однако его формы и идиомы работают против этого. Например, хотя в JavaScript удобно использовать глобальные переменные, из-за сложностей, связанных с областями видимости, это становится проблематично.

Я задействую единственную глобальную переменную для хранения приложения или библиотеки. Каждый объект имеет собственное пространство имен, поэтому, используя объекты, мне удается легко организовать свой код. А применение замыканий, обеспечивающих дополнительное скрытие информации, повышает надежность моих модулей.

10 Прекрасные черты

> В ожидании ответа оскверняю мои губы
> твоими стопами, глаза — твоим обликом,
> сердце — всеми частями твоего тела. Твой,
> в глубочайшей готовности к служению
> пребывающий...
>
> *Уильям Шекспир. Бесплодные усилия любви*

В прошлом году Энди Орам (Andy Oram) и Грег Уилсон (Greg Wilson) пригласили меня внести свой вклад в главу для их книги «Идеальный код» (O'Reilly), антологии на тему красоты в компьютерных программах. Я хотел написать свою главу на JavaScript. Представить этот язык как некий абстрактный, мощный и полезный инструмент, чтобы показать, на что он способен. Кроме того, я хотел избежать браузеров и других инструментов, где применение JavaScript тривиально. Мне хотелось показать нечто респектабельное и значимое.

Я сразу подумал об инструменте Вогана Пратта (Vaughn Pratt) для разбора выражений, который я использую в JSLint (см. приложение B). Разбор выражений — важная тема в вычислительной технике. Способность написать компилятор сама по себе является для языка испытанием на прочность.

Я хотел представить весь код парсера на JavaScript, причем программа должна была осуществлять синтаксический разбор JavaScript. Однако моя глава была лишь одной из 30 или 40, так что количество страниц было ограничено. Еще одна сложность состояла в том, что большинство потенциальных читателей моей главы не имело опыта работы с JavaScript, поэтому я должен был познакомить их с этим языком и его особенностями.

Итак, я решил описать подмножество языка, иначе мне пришлось бы разбирать весь язык JavaScript. Я назвал это подмножество Simplified JavaScript. Выбор подмножества не был трудным: оно включало только функции, необходимые для написания парсера. Вот как я описал его в книге «Идеальный код»:

> Simplified JavaScript — это только хорошее, в том числе:
>
> *Функции, как первый класс объектов*
>
> Функции в Simplified JavaScript — это лямбда-выражения внутри лексического контекста.

Динамические объекты с прототипизированным наследованием

Объекты не привязаны к классам. Можно добавить новый член к любому объекту обычным присваиванием. Объект может наследовать члены от другого объекта.

Литералы объектов и литералы массивов

С помощью литералов очень удобно создавать новые объекты и массивы. JavaScript-литералы стали источником вдохновения для формата обмена данными JSON.

В этой подборке перечислено все самое лучшее, что есть в моей книге. Даже если бы этот язык обладал только этими чертами, он был бы очень выразительным и мощным. В JavaScript есть множество дополнительных возможностей, которые дают не так уж много, но, как будет видно из приложений, имеют ряд негативных особенностей. А в этой моей подборке нет ничего опасного или плохого. Все только самое лучшее.

Simplified JavaScript не является строгим подмножеством JavaScript. Я добавил в него несколько новых функций. Добавил число `pi` как простую константу. Я сделал это, чтобы продемонстрировать особенности парсера. Кроме того, я применил улучшенную политику зарезервированных слов и показал, что они не нужны. В функции слово не может быть одновременно использовано и как переменная или имя параметра, и как характеристика языка. Вы можете задействовать его для чего-то одного, и у программиста появляется выбор. Это позволяет быстрее освоить язык, так как не обязательно быть в курсе его особенностей, которыми вы не пользуетесь. И это делает язык более расширяемым, поскольку нет необходимости резервировать слова для описания новых возможностей.

Кроме того, я добавил область видимости для блоков; это хотя и не является необходимым, но ее отсутствие смущает опытных программистов. Я включил область видимости для блоков, так как предполагал, что моя программа разбора будет использоваться для синтаксического разбора не только JavaScript, но других языков, область видимости которых задана правильно. Код своей программы я написал так, чтобы было совершенно не важно, доступны области видимости блоков или нет. Я рекомендую вам писать точно так же.

Когда я начал думать о своей книге, то хотел развить эту мою идею подмножества, позволяющую показать, как можно взять существующий язык программирования и существенно его улучшить, просто исключив из него некоторые малозначащие возможности.

Существует множество продуктов, где во главу угла ставятся те или иные характеристики, а цена, которую приходится за них платить, не учитывается. Это способно оказать негативное влияние на потребителей, поскольку может усложнить понимание и использование продукта. Людям нравятся продукты, которые работают просто. Оказывается, что создавать проекты, которые способны работать просто, намного сложнее, чем проекты, предлагающие огромный список возможностей.

Каждая дополнительная возможность имеет свою цену, связанную с затратами на спецификацию, проектирование и развертывание. Существуют затраты и на тестирование, и на поддержание надежности. Чем больше характеристик, тем больше трудностей в их разработке и взаимодействии. Незначительные затраты на хранение информации в стационарных программных комплексах становятся значительными в мобильных приложениях. Возрастают затраты на поддержание производительности, поскольку закон Мура не распространяется на батарейки.

Все характеристики связаны с затратами на их документирование. Описание каждой характеристики добавляется к страницам руководства, повышая затраты на обучение. Затраты на то, что необходимо лишь единичному числу пользователей, ложатся на плечи всех остальных. Однако при разработке программных продуктов и языков программирования нам хотелось бы иметь только то, что хорошо работает и реально востребовано, потому что только тогда мы сможем создавать нечто полезное.

Все находят что-то хорошее в продуктах, которыми пользуются. Все ценят простоту, а если нам не могут предложить эту простоту, мы пытаемся организовать ее сами. Моя микроволновая печь имеет множество возможностей, но я использую ее только для подогрева пищи. То же самое с часами, настройка которых — дело не простое. Мы справляемся с чрезмерной сложностью некоторых конструкций, находя и используя в них только то, что нам реально необходимо.

Вообще-то было бы неплохо, если бы программные продукты и языки программирования имели только достоинства.

A Кошмарные вещи

... и на словах
И в жизни помнит божий страх...

Уильям Шекспир. Перикл, царь Тирский

В этом приложении представлены проблемные черты JavaScript, избежать которых не так уж просто. Следует помнить о них, чтобы всегда быть готовым справиться с ситуацией.

Глобальные переменные

Самое кошмарное из всего плохого, что есть в JavaScript, связано с глобальными переменными. *Глобальная переменная* — это такая переменная, которая видна в любой части программы. Такие переменные могут быть полезны в небольших программах, но как только программы становятся больше, управлять глобальными переменными становится все труднее. Ведь глобальную переменную в любой части программы всегда можно изменить, а это может существенно сказаться на поведении программы. Использование глобальных переменных снижает надежность программ, в которых они применяются.

Глобальные переменные затрудняют запуск внутри программы независимых подпрограмм. Если глобальные переменные основной программы и подпрограмм будут иметь одинаковые имена, то они начнут мешать друг другу, но определить, в чем проблема, будет довольно трудно.

Глобальные переменные используются во многих языках. Например, открытые статические члены классов в Java являются глобальными переменными. Проблема JavaScript не только в том, что этот язык позволяет существовать глобальным переменным, но и в том, что он от них требует. В JavaScript нет компоновщика. Все единицы компиляции загружаются в общий глобальный объект.

Существуют три способа объявления глобальных переменных. Во-первых, такую переменную можно объявить вне функций с ключевым словом `var`:

```
var foo = value;
```

Во-вторых, глобальную переменную можно добавить непосредственно как свойство глобального объекта. Глобальный объект является контейнером для всех глобальных переменных. В веб-браузерах глобальный объект называется `window`:

```
window.foo = value;
```

В-третьих, можно использовать переменную без объявления, в этом случае *подразумевается*, что переменная является глобальной:

```
foo = value;
```

Необязательность объявления переменных перед их использованием может быть удобной для начинающих. Но, к сожалению, забыть объявить переменную — довольно распространенная ошибка. Политика JavaScript превращать необъявленные переменные в глобальные, что приводит к ошибкам, которые очень трудно найти.

Область видимости

Синтаксис JavaScript происходит от C. Во всех остальных C-подобных языках блок (набор инструкций, заключенных в фигурные скобки) создает область видимости. Переменные, объявленные внутри блока, не видны снаружи. Хотя в JavaScript используется блочный синтаксис, блоки имеют собственную область видимости: переменная, объявленная в блоке, видна внутри всей функции, содержащей блок. Это может неприятно удивить программистов, имеющих опыт работы с другими языками.

Как правило, объявлять переменные в большинстве языков лучше всего при их первом использовании, чего в JavaScript делать нежелательно именно потому, что блоки в JavaScript не имеют собственной области видимости. Поэтому объявлять все переменные следует в начале каждой функции.

Автоматическая вставка точки с запятой

В JavaScript существует механизм, который пытается дополнять строки точками с запятой, автоматически корректируя распространенную ошибку программирования. Однако не стоит на него надеяться. Этот механизм может спровоцировать более серьезные ошибки.

Иногда он вставляет точку с запятой там, где это не требуется. Рассмотрим возможные последствия его работы на примере инструкции `return`.

Если инструкция `return`:

```
return
{
    status: true
};
```

Подразумевается, что будет возвращен объект, имеющий член `status`. К сожалению, вставка точки с запятой сразу после ключевого слова `return` превращает его в инструкцию, возвращающую значение `undefined`. Нигде не говорится, что автоматическая вставка точки с запятой может привести к неправильной интерпретации программы. Избежать этой проблемы можно, если взять за правило ставить открывающую фигурную скобку в конце строки после ключевого слова `return`, а не в начале следующей строки:

```
return {
    status: true
};
```

Зарезервированные слова

В JavaScript зарезервированы следующие слова:

```
abstract boolean break byte case catch char class const continue
debugger default delete do double else enum export extends false final
finally float for function goto if implements import in instanceof int
interface long native new null package private protected public return
short static super switch synchronized this throw throws transient true
try typeof var volatile void while with
```

При этом большинство из них в самом языке JavaScript не используется.

Ключевыми словами нельзя называть переменные и параметры. Когда необходимо использовать зарезервированные слова в качестве ключей в литералах объектов, их следует заключать в кавычки. Их нельзя указывать с точкой, поэтому иногда вместо этого приходится добавлять скобки:

```
var method;                 // допустимо
var class;                  // недопустимо
object = {box: value};      // допустимо
object = {case: value};     // недопустимо
object = {'case': value};   // допустимо
object.box = value;         // допустимо
object.case = value;        // недопустимо
object['case'] = value;     // допустимо
```

Unicode

Язык JavaScript разрабатывался в то время, когда стандарт Unicode описывал не более 65 536 символов. С тех пор его объем возрос более чем до одного миллиона символов.

В JavaScript используются 16-разрядные символы. Этого достаточно, чтобы покрыть первоначальный объем в 65 536 символов, который сейчас известен как основная многоязыковая плоскость (basic multilingual plane). Каждый из оставшихся символов можно представить с помощью пары символов. Проблема в том, что такие пары в Unicode считаются одним символом, а в JavaScript — двумя разными символами.

typeof

Оператор `typeof` возвращает строку, показывающую тип операнда. Например:

```
typeof 98,6
```

Это выражение возвращает `'number'`. Еще пример:

```
typeof null
```

К сожалению, это выражение возвращает `'object'`, а не `'null'`. Увы. Лучше проверять `null` попроще:

```
my_value === null
```

Еще большей проблемой является проверка «объектности». Оператор `typeof` не может отличить `null` от объекта, но проверку можно выполнить иначе, поскольку значение `null` является ложным, а все объекты истинны:

```
if (my_value && typeof my_value === 'object') {
    // my_value – это объект или массив!
}
```

Мы еще вернемся к этой теме далее в разделах «NaN» и «Странные массивы».

В различных реализациях тип объектов регулярных выражений трактуется поразному. Например:

```
typeof /a/
```

В некоторых реализациях это выражение возвращает `'object'`, в других — `'function'`. Было бы полезней, если бы возвращалось `'regexp'`, но стандарты этого не позволяют.

parseInt

Функция `parseInt` просто преобразует строку в целое число. Однако `parseInt("16")` и `parseInt("16 tons")` дадут одинаковый результат. Было бы неплохо, если бы функция как-нибудь сообщала о дополнительном тексте, но этого не происходит.

Если первый символ строки равен 0, то строка преобразуется в число с основанием 8, а не с основанием 10. В системе с основанием 8 нет цифр 8 и 9, поэтому вызовы `parseInt("08")` и `parseInt("09")` возвращают 0. Эта ошибка приводит к проблемам в программах, анализирующих дату и время. К счастью, в `parseInt` систему счисления можно задать с помощью параметра, так что вызов `parseInt("08", 10)` возвращает 8. Рекомендую всегда задавать этот параметр.

Оператор +

Оператор `+` может использоваться и для сложения, и для конкатенации строк. Как именно он будет работать, зависит от типа параметров. Если один из операндов является пустой строкой, другой операнд преобразуется в строку. Если оба операнда являются числами, оператор `+` возвращает их сумму. В противном случае он преобразует оба операнда в строки и выполняет конкатенацию. Такое сложное поведение является распространенным источником ошибок. Если вы собираетесь произвести сложение, убедитесь, что оба операнда — числа.

Плавающая точка

Двоичные числа с плавающей точкой не работают с десятичными дробями, например: `0.1 + 0.2` не равно `0.3` — это наиболее распространенная ошибка в JavaScript, которая является следствием применения стандарта IEEE, принятого для двоичной арифметики с плавающей точкой (IEEE 754). Этот стандарт идеально подходит для многих приложений, но нарушает привычное представление о числах, заложенное всем в средней школе.

К счастью, целочисленная арифметика с плавающей точкой дает точные результаты, так что ошибок десятичного представления можно избежать путем округления.

Например, долларовые значения умножением на 100 могут быть преобразованы в целочисленные значения центов. Центы складываются нормально. Затем сумму можно разделить на 100 и преобразовать обратно в доллары. Когда люди считают деньги, они хотят, чтобы результаты были точными.

NaN

Значение `NaN` — это специальная величина, определенная стандартом IEEE 754. Она означает не число, и тем не менее:

```
typeof NaN === 'number' // true
```

Значение NaN можно получить при попытке преобразовать строку в число, если строка не похожа на число. Например:

```
+ '0'        // 0
+ 'oops'     // NaN
```

Если NaN выступает в роли операнда в арифметическом выражении, то результатом тоже является NaN. Так что если цепочка формул возвращает результат NaN, по крайней мере, одним из входных параметров было значение NaN или оно было получено как промежуточный результат.

Можно устроить для NaN проверку. Как говорилось ранее, оператор typeof не видит разницы между 'number' и NaN, к тому же оказывается, что NaN не равно самому себе. На удивление:

```
NaN === NaN        // false
NaN !== NaN        // true
```

Чтобы отличить 'number' и NaN, в JavaScript используется функция isNaN:

```
isNaN(NaN)         // true
isNaN(0)           // false
isNaN('oops')      // true
isNaN('0')         // false
```

Функция isFinite — это лучшее средство определить, может ли переменная быть использована как число, потому что она отсеет значения NaN и Infinity. К сожалению, isFinite будет пытаться конвертировать операнд в число, так что это лучший вариант, если переменная на самом деле не является числом. Можно определить собственную функцию isNumber:

```
var isNumber = function isNumber(value) {
    return typeof value === 'number' && isFinite(value);
}
```

Странные массивы

В JavaScript нет настоящих массивов. Но не все так плохо. Массивы в JavaScript предельно просты в использовании. Не нужно задавать их размерность, и они никогда не вызовут ошибок, связанных с выходом за границы массива. Хотя они и не так эффективны, как настоящие массивы.

Оператор typeof не делает различий между массивами и объектами. Для того чтобы определить, является ли значение переменной массивом, необходимо получить значение ее свойства constructor:

```
if (my_value && typeof my_value === 'object' &&
        my_value.constructor === Array) {
    // my_value – это массив!
}
```

Такая проверка даст отрицательный результат, если массив был создан в другом фрейме или в другом окне. Вот реализация более надежной проверке значения переменной, созданной в другом фрейме:

```
if (Object.prototype.toString.apply(my_value) === '[object Array]'){
    // my_value - это действительно массив!
}
```

Массив аргументов — это не массив, а объект, имеющий член `length` (длина массива). Подобные проверки не покажут, что массив аргументов является массивом.

Значения falsy

JavaScript обладает удивительно большим набором значений `falsy`, которые представлены в табл. А.1.

Таблица А.1. Многочисленные falsy-значения в JavaScript

Значение	Тип
`0`	`Number`
`NaN` (не число)	`Number`
`' '` (пустая строка)	`String`
`false`	`Boolean`
`null`	`Object`
`undefined`	`Undefined`

Хотя все эти значения являются ложными, они не взаимозаменяемы. Вот, к примеру, неудачный способ определить, является ли объект отсутствующим членом:

```
value = myObject[name];
if (value == null) {
    alert(name + ' not found.');
}
```

Хотя значением отсутствующих членов объекта является `undefined`, здесь проводится проверка на `null`. К тому же здесь вместо более надежного оператора `===` используется оператор `==` (детали см. в приложении Б), который производит сравнение с приведением типов. Иногда две эти ошибки компенсируют друг друга, иногда нет.

Значения `undefined` и `NaN` не константы — это глобальные переменные, значения которых можно изменить. Так не должно быть, но так есть. И лучше этого не делать.

hasOwnProperty

В главе 3, чтобы обойти проблемы, связанные с инструкцией `for in`, предлагалось использовать в качестве фильтра метод `hasOwnProperty`. К сожалению, `hasOwnProperty` — это метод, а не оператор, поэтому в любом объекте он может быть заменен другой функцией или даже не функцией, а просто значением:

```
var name;
another_stooge.hasOwnProperty = null; // проблема
for (name in another_stooge) {
    if (another_stooge.hasOwnProperty(name)) { // бум
        document.writeln(name + ': ' + another_stooge[name]);
    }
}
```

Object

Объекты в JavaScript никогда реально не бывают пустыми, потому что они получают свои члены из цепочки прототипов. Иногда это очень важно. К примеру, предположим, что вы пишете программу, подсчитывающую количество вхождений каждого слова в текст. Можно использовать метод `toLowerCase` для приведения текста к нижнему регистру, а затем для получения массива слов использовать метод регулярных выражений `split`. После этого можно перебрать все слова и посчитать, сколько раз встречается каждое из них:

```
var i;
var word;
var text =
        "This oracle of comfort has so pleased me, " +
        "That when I am in heaven I shall desire " +
        "To see what this child does, " +
        "and praise my Constructor.";

var words = text.toLowerCase().split(/[\s,.]+/);
var count = {};
for (i = 0; i < words.length; i += 1) {
    word = words[i];
    if (count[word]) {
        count[word] += 1;
    } else {
        count[word] = 1;
    }
}
```

Если мы посмотрим на результаты, то выясним, что `count['this']` равно 2, `count.heaven` равно 1, а вот в `count.constructor` окажется сумасшедшего вида

строка. Причина в том, что объект `count` наследуется от `Object.prototype`, а `Object.prototype` содержит элемент с именем `constructor`, значением которого является `Object`. Оператор `+=`, как и оператор `+`, не производит сложение, если его операнды не являются числами. `Object` — это функция, поэтому `+=` каким-то непонятным образом преобразует ее в строку и выполняет ее конкатенацию с единицей.

Можно избежать проблем, напоминающих проблему инструкции `for in`, используя метод `hasOwnProperty` или подбирая конкретные типы. В этом случае проверка истинности `count[word]` не совсем ясна. Вместо нее можно было бы написать:

```
if (typeof count[word] === 'number') {
```

Б Плохие вещи

...так я объявлю его трусом. Скажи мне,
за какой из моих пороков полюбила ты меня?

Уильям Шекспир. Много шума из ничего

В этом приложении я описываю некоторые недостатки JavaScript, которые можно легко обойти. Действуя в этом направлении, можно совершенствовать язык JavaScript и совершенствоваться как программисту.

Оператор ==

В JavaScript есть две группы операторов равенства/неравенства, это операторы `===` и `!==` и их злые близнецы `==` и `!=`. Хорошие работают так, как и следовало ожидать. Если два операнда одного и того же типа имеют одинаковое значение, то оператор `===` возвращает значение `true`, а оператор `!==` возвращает `false`. Их злые близнецы тоже делают все правильно, если оба операнды одного типа, но если нет, они пытаются привести значения к одному типу. Правила, по которым они это делают, очень запутаны и их непросто запомнить. Вот лишь некоторые интересные случаи:

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true

false == undefined  // false
false == null       // false
null == undefined   // true

' \t\r\n ' == 0     // true
```

Нетранзитивность вызывает опасения. Советую всегда использовать только операторы `===` и `!==` и никогда не использовать их злых близнецов. Все только что приведенные примеры при сравнении с помощью оператора `===` возвращают значение `false`.

Инструкция with

Инструкция `with` в JavaScript должна обеспечивать быстрый доступ к свойствам объекта. К сожалению, результаты иногда могут быть непредсказуемы, поэтому ее использования следует избегать.

Пример:

```
with (obj) {
    a = b;
}
```

Эта инструкция делает то же самое, что и следующая:

```
if (obj.a === undefined) {
    a = obj.b === undefined ? b : obj.b;
} else {
    obj.a = obj.b === undefined ? b : obj.b;
}
```

Таким образом, в результате выполняется одна из следующих инструкций:

```
a = b;
a = obj.b;
obj.a = b;
obj.a = obj.b;
```

Во время чтения программы невозможно сказать, какая из этих инструкций будет получена. Ситуация может меняться при каждом запуске программы или даже в ходе ее выполнения. Если вам не удается прочитать программу и понять, что она собирается делать, нельзя быть уверенным, что все будет сделано правильно.

Просто присутствуя в языке, инструкция `with` заметно тормозит процессор JavaScript, поскольку нарушает лексические связи между именами переменных. Намерения были благими, но языку было бы лучше без нее.

eval

Функция `eval` передает строку компилятору JavaScript и обрабатывает результат. Многие злоупотребляют этой функцией. Чаще всего ее используют те, кто не

очень хорошо знает JavaScript. Например, если вы знаете о точечной нотации, но не знаете правил задания индекса, вы можете написать:

```
eval("myValue = myObject." + myKey + ";");
```

А надо было действовать проще:

```
myvalue = myObject[myKey];
```

То, что написано в инструкции с функцией `eval`, понять гораздо труднее. Да и выполняется эта инструкция значительно медленнее, потому что приходится запускать компилятор, чтобы обработать тривиальное присваивание. Кроме того, она мешает верификатору JSLint (детали см. в приложении В) в плане выявления проблем в коде.

Кроме того, функция `eval` ставит под угрозу безопасность разрабатываемого приложения, поскольку предоставляет слишком много полномочий тексту внутри функции. А это подобно инструкции `with` влияет на язык в целом.

Конструктор `Function` — еще одна форма функции `eval` и его также следует избегать.

Браузер предоставляет функции `setTimeout` и `setInterval`, принимающие в качестве аргументов строки или функции. Если аргументы являются строками, `setTimeout` и `setInterval` работают аналогично функции `eval`. Аргументов в строковой форме также следует избегать.

Инструкция continue

Инструкция `continue` передает управление в начало цикла. Я еще никогда не видел такого фрагмента кода, который не стал бы лучше после того, как из него убрали все инструкции `continue`.

Провал сквозь switch

Инструкция `switch` напоминает `go to` из FORTRAN IV. Если явно не прерывать течение программы, после выполнения каждой ветви `case` будет происходить переход к следующей ветви `case`.

Кто-то писал мне, что верификатор JSLint должен предупреждать, если происходит подобный провал с одной ветви `case` на другую. Он отмечал, что это довольно распространенная ошибка, которую трудно заметить в коде. Я ответил, что все верно, но получаемая компактность кода с лихвой компенсирует вероятность ошибки.

На следующий день он сообщил, что JSLint сигнализирует об ошибке, причем ошибка определяется неверно. Я все проверил и выяснил, что это был как раз такой случай провала сквозь инструкцию `switch`. И тогда меня осенило. Теперь я

больше не пытаюсь специально организовать подобные провалы, поскольку тогда проще найти провал, если он произойдет по ошибке.

Плохим в языке является не то, что явно опасно или бесполезно. Этим можно просто не пользоваться. Плохими являются кажущиеся привлекательными возможности, которые одновременно и полезны, и опасны.

Безблочные инструкции

Инструкции `if`, `while`, `do` или `for` могут быть блочными или безблочными. Безблочная форма этих инструкций — еще один очевидный недостаток JavaScript. Она позволяет сэкономить два символа, но это — сомнительное достоинство. Безблочная форма скрывает структуру программы, что при последующих манипуляциях с кодом может легко породить ошибки. Например:

```
if (ok)
    t = true;
```

Эта инструкция при модификации может стать такой:

```
if (ok)
    t = true;
    advance();
```

А это напоминает следующее:

```
if (ok) {
    t = true;
    advance();
}
```

Хотя на самом деле означает совсем другое:

```
if (ok) {
    t = true;
}
advance();
```

Программы, которые должны делать одно, но на самом деле делают другое, вряд ли можно считать правильными. Постоянное аккуратное использование блоков делает все проще и понятнее.

Операторы ++ и --

Операторы инкремента и декремента делают код чрезвычайно кратким. В таких языках, как C, это дает возможность одной строкой кода задать весь процесс копирования строки:

```
for (p = src, q = dest; !*p; p++, q++) *q = *p;
```

Так поощряется довольно опрометчивый стиль программирования. Большинство ошибок переполнения буфера, порождающих кошмарные уязвимости, связано с подобным кодом.

На собственной практике я заметил, что код при наличии операторов `++` и `--`, как правило, становился слишком угловатым, сложным и загадочным. Так что из соображений дисциплины я их больше не использую. Думаю, что в результате мой стиль кодирования стал чище.

Битовые операторы

В JavaScript имеется тот же набор битовых операторов, что и в Java:

- `&` — и;
- `|` — или;
- `^` — исключающее или;
- `~` — не;
- `>>` — правый сдвиг с переносом знака;
- `>>>` — правый сдвиг без переноса знака;
- `<<` — левый сдвиг.

В Java битовые операторы работают с целыми числами. В JavaScript целых чисел нет, а есть только вещественные числа с плавающей точкой. Таким образом, битовые операторы преобразуют числа-операнды в целые числа, и проделав с ними все необходимые операции, преобразуют их обратно. В большинстве языков такие операторы близки к аппаратным и довольно быстры. В JavaScript они очень далеки от аппаратных и чрезвычайно медленны. Из-за этого битовые манипуляции в JavaScript используется крайне редко.

То есть, вероятнее всего, в программах на JavaScript оператор `&` — это просто устаревший оператор `&&`. Наличие битовых операторов в некоторой степени уменьшает избыточность языка, но позволяет ошибкам оставаться незамеченными.

Инструкция function против выражения function

В JavaScript есть как инструкция `function`, так и выражение `function`. Это сбивает с толку, так как они могут выглядеть совершенно одинаково. Инструкция `function` — это сокращенная форма инструкции `var` со значением функции.

Например:

```
function foo() {}
```

Эта инструкция значит то же, что и следующая:

```
var foo = function foo() {};
```

В этой книге я использую вторую форму написания, поскольку она дает понять, что `foo` — это переменная, содержащая значение функции. Чтобы использовать язык правильно, важно понимать, что функции являются значениями.

Инструкции `function` поднимаются *вверх*. Это означает, что независимо от того, где находится инструкция `function`, она перемещается в верхнюю часть области видимости, в которой она определена. Это смягчает требование о том, что функции должны объявляться перед своим использованием, что открывает лазейку для небрежности. Кроме того, это мешает использованию инструкций `function` внутри инструкций `if`. И это при том, что большинство браузеров допускают наличие инструкций `function` внутри инструкций `if`, но интерпретируют эту ситуацию по-разному, что приводит к проблемам переносимости.

Первым элементом в инструкции не может быть выражение `function`, так как официальная грамматика предполагает, что инструкция, которая начинается со слова `function`, является инструкцией `function`. Обойти это ограничение можно, заключив весь вызов в скобки:

```
(function ( ) {
    var hidden_variable;

    // Эта функция влияет на свое
    // окружение, но не вводит новых
    // глобальных переменных.
}());
```

Типизированные оболочки

JavaScript содержит набор типизированных оболочек, например:

```
new Boolean(false)
```

Эта инструкция возвращает объект, имеющий метод `valueOf`, который возвращает обернутое значение. Это не только совершенно не нужно, но и непонятно. Никогда не используйте `new Boolean`, `new Number` или `new String`.

Кроме того, избегайте `new Object` и `new Array`, используйте вместо них скобки `{}` и `[]`.

Оператор new

Оператор `new` в JavaScript создает новый объект, который наследуется от члена прототипа операнда, а затем вызывает операнд, связывая новый объект с `this`. Это

дает операнду (лучше, чтобы это был конструктор функции) возможность задать новый объект, прежде чем будет обработан запрос.

Если вы забыли использовать оператор `new`, вместо него вы получите обычный вызов функции, и свойство `this` будет связано не с новым, а с глобальным объектом. Значит, что при попытке инициализации новых членов функция будет затирать глобальные переменные. Это очень плохо. Тем более учитывая, что никаких предупреждений ни во время компиляции, ни во время выполнения не возникнет.

Функциям, предназначенным для использования с оператором `new`, принято давать имена, начинающиеся с прописной буквы. А имена, начинающиеся с прописной буквы, должны применяться только с конструктором функции, который получает префикс `new`. Это соглашение дает визуальный сигнал, указывающий на место возникновения ошибок, которые сам язык стремится не замечать.

Поэтому лучше всего вовсе не использовать оператор `new`.

Оператор void

Во многих языках `void` — это тип, который не имеет значения. В JavaScript `void` является оператором, принимающим операнд и возвращающим неопределенное значение. Оператор `void` — весьма бесполезная и запутанная конструкция, которой лучше избегать.

B JSLint

... Какое заблужденье
Пристало к зрению и слуху моему?

Уильям Шекспир. Комедия ошибок

Когда C был еще молодым языком программирования, довольно часто возникали программные ошибки, которые не выявлялись примитивными компиляторами, поэтому была разработана вспомогательная программа lint, сканирующая исходный файл в поисках проблем.

Когда язык C заматерел, был определен точнее, его надежность возросла, компиляторы стали выдавать более точные предупреждения, и потребность в lint пропала.

JavaScript — довольно молодой язык программирования. Первоначально он был ориентирован на решение достаточно простых задач в веб-страницах, для которых язык Java был слишком сложным и неудобным. Однако новый язык оказался способен на большее и в настоящее время он широко используется даже в больших проектах. Однако многие возможности языка, призванные упростить его применение, в больших проектах могут стать источником серьезных проблем. То есть для JavaScript потребовалась своя программа lint, которой стала JSLint — средство верификации и проверки синтаксиса JavaScript.

JSLint — это инструмент, проверяющий качество JavaScript-кода: на входе он получает исходный текст и сканирует его в поисках проблем. Если JSLint обнаруживает проблему, он возвращает сообщение, содержащее ее описание и приблизительное местоположение. Проблема — это не обязательно синтаксическая ошибка, хотя часто именно она. JSLint проверяет также стилистические соглашения и выявляет структурные проблемы. Верификатор JSLint не гарантирует, что ваша программа верна, он просто дает шанс взглянуть свежим взглядом на возможные проблемы.

JSLint определяет специальное подмножество JavaScript, более строгое чем то, которое описано в третьем издании ECMAScript Language Specification. Это

подмножество тесно связано с рекомендациями относительно стиля программирования, которые я дал в главе 9.

JavaScript — достаточно сырой язык, но по сути он элегантен и хорош. JSLint помогает сделать ваши программы грамотнее, избежать множества неприятностей.

JSLint можно найти по адресу http://www.JSLint.com/.

Неопределенные переменные и функции

Самая серьезная проблема JavaScript — глобальные переменные. Если переменная не объявлена явно (как правило, с помощью инструкции `var`), то JavaScript считает ее глобальной. А это может скрывать ошибки в именовании и ряд других проблем.

JSLint считает, что все переменные и функции должны объявляться перед тем, как будут использованы или вызваны, что позволяет выявлять подразумеваемые глобальные переменные. Такая практика помогает делать программы более понятными.

Иногда в файле используются глобальные переменные и функции, определенные в другом месте. Можно сообщить это JSLint, включив в файл комментарии со списком глобальных функций и объектов, используемых вашей программой, но не определенных в ней или в файле сценария.

В комментариях с глобальными объявлениями могут быть перечислены все имена, которые вы намеренно используете в качестве глобальных переменных. JSLint может задействовать эту информацию для выявления пропущенных и забытых объявлений `var`. Глобальное объявление может выглядеть следующим образом:

```
/*global getElementByAttribute, breakCycles, hanoi */
```

Глобальное объявление начинается с `/*global`. Обратите внимание, что перед буквой `g` нет пробела. Можно создавать сколько угодно комментариев, начинающихся с символов `/*global`. Они должны быть указаны прежде, чем будут использованы перечисленные в них переменные.

Некоторые глобальные переменные могут предопределяться (детали см. в разделе «Опции»). Выбор варианта Assume a browser (Предполагаемый браузер) предопределяет стандартные глобальные свойства веб-браузеров, такие как `window`, `document` и `alert`. Выбор варианта Assume Rhino (Предполагаемая среда Rhino) заранее определяет глобальные свойства среды Rhino. Выбор варианта Assume a Yahoo Widget (Предполагаемый Yahoo-виджет) заранее определяет глобальные свойства среды Yahoo-виджетов.

Members

Так как JavaScript — динамический объектно-ориентированный язык со слабой типизацией, во время компиляции невозможно определить, есть ли ошибки в написании имен свойств. JSLint может в этом помочь.

В конце своего отчета JSLint помещает комментарий `/*members*/`, содержащий все имена и строковые литералы, которые используются для именования членов объектов в точечной нотации, в индексной нотации или в литералах объектов. Это позволяет просмотреть список опечаток. Имена членов, встречающихся только один раз, выделяются курсивом, поэтому обнаружить опечатки гораздо легче.

Можно скопировать комментарий `/*members*/` в файл сценария. JSLint по списку проверит написание всех имен свойств. Таким образом, можно использовать JSLint для поиска опечаток:

```
/*members doTell, iDoDeclare, mercySakes,
    myGoodness, ohGoOn, wellShutMyMouth */
```

Опции

Реализация JSLint получает особый объект опций, что позволяет вам настроить требуемое подмножество JavaScript. Кроме того, опции можно задать непосредственно в сценарии.

Задание опций может выглядеть следующим образом:

```
/*jsLint nomen: true, evil: false */
```

Спецификация начинается с символов `/*jsLint`. Обратите внимание, что перед символом `j` нет пробела. Подобная спецификация содержит последовательность пар имя/значение, где имена — это JSLint-опции, а значения — `true` или `false`. Приоритет спецификации опций выше, чем приоритет объекта опций. Все опции по умолчанию имеют значение `false`. Опции, доступные в JSLint, перечислены в табл. B.1.

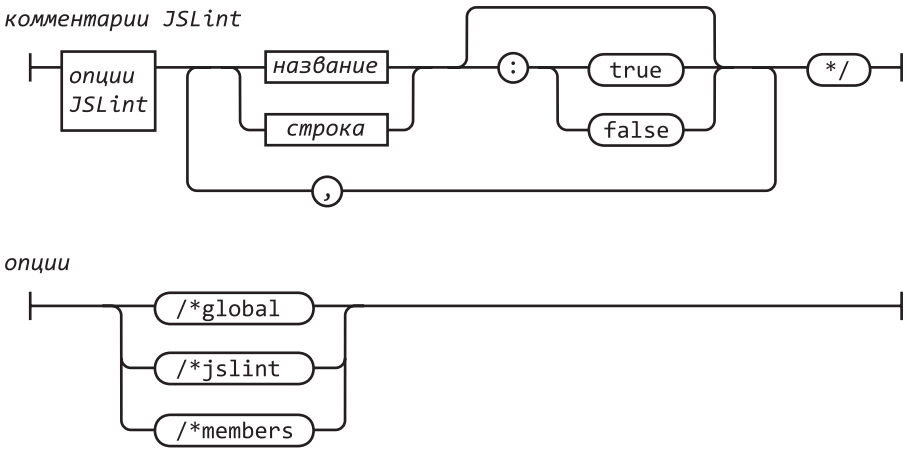
Таблица B.1. Доступные в JSLint опции

Опция	Значение
adsafe	true, если применяются правила ADsafe.org
bitwise	true, если допускается использование битовых операторов
browser	true, если предопределены стандартные глобальные свойства веб-браузеров
cap	true, если в HTML допускается верхний регистр
debug	true, если допускается использование отладки
eqeqeq	true, если допускается использование оператора ===
evil	true, если допускается использование функции eval
forin	true, если допускается использование инструкции for in без фильтра
fragment	true, если допускается включение HTML-фрагментов
laxbreak	true, если требуется проверять инструкции прерывания

продолжение ⇩

Таблица B.1 (*продолжение*)

Опция	Значение
nomen	true, если требуется проверять имена
on	true, если допускается применение HTML-обработчиков событий
passfail	true, если сканирование должно быть прекращено при обнаружении первой ошибки
plusplus	true, если не допускается применение операторов ++ и —
rhino	true, если заранее определены глобальные свойства среды Rhino
undef	true, если неопределенные глобальные переменные считаются ошибкой
white	true, если применяются строгие правила расстановки пробелов
widget	true, если заранее определяет глобальные свойства Yahoo-виджетов



Точка с запятой

В JavaScript используется C-подобный синтаксис, требующий символа точки с запятой для разделения выражений. Разработчики JavaScript попытались сделать так, чтобы программисту не нужно было самому вводить точки с запятой, предусмотрев механизм их автоматической вставки. Однако это опасно.

Как и в C, в JavaScript есть операторы `++`, `--` и `(`, которые могут быть как префиксами, так и суффиксами. А точка с запятой может привести к неоднозначности выражения.

В JavaScript разрыв строки может расцениваться и как пробел, и как место для точки с запятой. Вместо одной неопределенности возникает другая.

JSLint считает, что за каждой инструкцией должен располагаться знак точки с запятой (`;`), исключая инструкции `for`, `function`, `if`, `switch`, `try` и `while`. JSLint не предполагает возможности существования ненужных точек с запятой или пустых инструкций.

Разрыв строки

Для того чтобы не пропустить ошибки, связанные с механизмом автоматического дополнения строк точкой с запятой, JSLint предполагает, что длинная инструкция может прерваться только после одного из следующих знаков препинания или операторов:

```
, ; : { } ( [ = < > ? ! + - * / % ~ ^ | &
== != <= >= += -=*= /= %= ^= |= &= << >> || &&
=== !== <<= >>= >>> >>>=
```

JSLint не ожидает, что длинная инструкция может прерваться после идентификатора, строки, числа, замыкания или любого из следующих суффиксных операторов:

```
) ] . ++ --
```

JSLint позволяет включить режим Tolerate sloppy line breaking *вид_разрыва* (Допускается случайный разрыв строки *вид_разрыва*).

Из-за автоматической вставки точек с запятой возможна маскировка ошибок копирования/вставки. Если вы привыкли завершать инструкцию переходом на новую строку, JSLint может помочь искать такие ошибки.

Запятая

Оператор запятой может чрезмерно усложнить выражение, кроме того, он маскирует некоторые ошибки программирования.

JSLint считает, что запятая может использоваться в качестве разделителя, но не оператора (исключая инициализирующую и инкрементирующую части инструкции `for`). JSLint не предполагает, что с помощью запятых могут обозначаться недостающие элементы литералов массивов. Дополнительные запятые не нужны. Не нужна запятая и после последнего элемента литерала массива или литерала объекта, поскольку некоторые браузеры могут неверно ее истолковать.

Обязательные блоки

JSLint предполагает, что инструкции `if` и `for` должны быть блочными, то есть содержать выражения, заключенные в фигурные скобки.

JavaScript позволяет записать инструкцию `if` следующим образом:

```
if (условие)
    инструкция;
```

Такая форма, как известно, может привести к ошибке, если над одним проектом трудится несколько программистов. Вот почему JSLint предполагает обязательное использование блоков:

```
if (условие) {
    инструкции;
}
```

Опыт показывает, что такая форма записи более надежна.

Закрытые блоки

Во многих языках блок вводит собственную область видимости. Переменные, указанные внутри блока, не видны снаружи блока.

Напротив, в JavaScript блоки не имеют собственной области видимости. Только функция обладает областью видимости. Переменная, введенная в любой точке функции, видна в любом месте функции. Блоки в JavaScript могут запутать опытных программистов, что ведет к ошибкам, потому что знакомый синтаксис порождает ложные ожидания.

JSLint считает, что блоки могут использоваться только с инструкциями `function`, `if`, `switch`, `while`, `for`, `do`, `try` и нигде больше. Исключение может быть сделано только для ветви `else` инструкции `if`, а также для инструкции `for in`.

Инструкция с выражением

Предполагается, что инструкция с выражением может быть присваиванием, вызовом функции/метода или вызовом оператора `delete`. Все остальные инструкции с выражением считаются ошибками.

Инструкция for in

Инструкция `for in` позволяет перебрать имена всех свойств объекта. К сожалению, она также перебирает еще и все те свойства, которые унаследованы по цепочке прототипов. Если вас интересуют только члены данных, этот механизм становится отрицательным побочным эффектом.

Тело каждой инструкции `for in` следует упаковать в инструкцию `if`, действующую как фильтр. Инструкция `if` позволяет выбрать значения определенного типа

или диапазон значений — это дает возможность исключить функции или исключить свойства, унаследованные от прототипов. Пример:

```
for (name in object) {
    if (object.hasOwnProperty(name)) {
        ...
    }
}
```

Инструкция switch

Распространенной ошибкой при написании инструкции `switch` является пропущенная инструкция `break` в конце каждой ветви `case`, что приводит к непреднамеренному «провалу» сквозь `switch`. JSLint считает, что перед каждой ветвью `case` и перед ветвью `default` должна находиться одна из инструкций передачи управления: `break`, `return` или `throw`.

Инструкция var

JavaScript позволяет указывать определение `var` в любом месте функции. JSLint подходит к этому строже.

JSLint ожидает, что:

- переменная объявляется с помощью `var` только один раз, причем до ее использования;
- функция объявляется до ее использования;
- параметры не объявляются как `var`.

JSLint не ожидает, что:

- массив аргументов объявляется как `var`;
- переменная объявляется в блоке (причина в том, что в JavaScript блоки не имеют собственной области видимости, что может иметь неожиданные последствия, так что лучше определять все переменные в верхней части тела функции.

Инструкция with

Инструкция `with` призвана обеспечивать быстрый доступ к членам вложенных объектов. Однако, к сожалению, при создании новых членов объекта она ведет себя очень странно. Вместо нее всегда используйте инструкцию `var`.

JSLint не предполагает возможность использования инструкции `with`.

Оператор =

JSLint не ожидает встретить оператор присваивания в условной части инструкции `if` или `while`. Пример:

```
if (a = b) {
    ...
}
```

JSLint считает это ошибкой, поскольку под этим, вероятнее всего, подразумевается следующее:

```
if (a == b) {
    ...
}
```

Операторы == и !=

Операторы `==` и `!=` перед сравнением выполняют приведение типов. Это плохо, потому что сравнение `'\f\r \n\t' == 0` вернет `true`. Из-за этого ошибки типизации могут остаться незамеченными.

При сравнении с любым из следующих значений всегда используйте операторы `===` или `!==`, не выполняющие приведения типов:

```
0 ' ' undefined null false true
```

Если приведение типов все же необходимо, используйте сокращенную форму записи. Пример:

```
(foo != 0)
```

Вместо этого пишите:

```
(foo)
```

Еще пример:

```
(foo == 0)
```

Вместо этого пишите:

```
(!foo)
```

Старайтесь всегда использовать операторы `===` и `!==`. Существует режим Disallow == and != (Запретить == и !=), который требует обязательного использования операторов `===` и `!==`.

Метки

В JavaScript любая инструкция может иметь метку; для меток существует даже отдельное пространство имен. JSLint относится к этому строже.

JSLint ожидает, что метки могут иметь только инструкции, взаимодействующие с инструкцией `break`. К таковым относятся `switch`, `while`, `do` и `for`. JSLint предполагает, что метки будут отличаться от переменных и параметров.

Недостижимый код

JSLint ожидает, что за инструкцией `return`, `break`, `continue` или `throw` всегда следует либо закрывающая фигурная скобка (`}`), либо ветвь `case` или `default`.

Путаница с плюсами и минусами

JSLint считает, что за оператором `+` не может следовать оператор `+` или `++`, а за оператором `-` не может следовать оператор `-` или `--`. В противном случае забытый пробел мог бы превратить `+ +` в `++`, и такую ошибку было бы довольно трудно обнаружить. Используйте скобки, чтобы избежать путаницы.

Инкремент и декремент

Операторы инкремента (`++`) и декремента (`--`), как известно, портят код, чрезмерно его усложняя. Хуже них могут быть только дефекты архитектуры, дающие дорогу вирусам, и другие бреши в системе безопасности. В JSLint режим использования этих операторов позволяет включить опция `plusplus`.

Битовые операторы

В JavaScript нет целого типа, но есть битовые операторы. Из-за отсутствия целого типа битовые операторы преобразуют операнды с плавающей точкой в целые числа и обратно, поэтому они не так эффективны, как в C или других языках. Битовые операторы редко используются в приложениях браузера. Их сходство с логическими операторами может спровоцировать ошибки. Опция `bitwise` запрещает использование этих операторов.

Зловещая функция eval

Функция `eval` и ее родственники (`Function`, `setTimeout` и `setInterval`) предоставляют доступ к компилятору JavaScript. Иногда это бывает полезно, но чаще свидетельствует о низком качестве кода. Функцией `eval` в JavaScript злоупотребляют чаще всего.

Оператор void

В большинстве C-подобных языков `void` — это тип. В JavaScript `void` является префиксным оператором, который всегда возвращает значение `undefined`. JSLint не ожидает встретить `void`, поскольку это сбивает с толку и абсолютно бесполезно.

Регулярные выражения

Регулярные выражения записываются в лаконичной и загадочной форме. JSLint ищет проблемы, которые могут спровоцировать ошибки переносимости. Кроме того, верификатор пытается определить двусмысленные фрагменты и рекомендует от них избавиться.

Синтаксис литералов регулярных выражений в JavaScript переопределяет символ `/`. Чтобы избежать двусмысленности, JSLint считает, что регулярному выражению предшествует символ `(`, `=`, `:` или `,`.

Конструкторы и префикс new

Конструкторы — это функции, предназначенные для использования с префиксом `new`. Префикс `new` создает новый объект на основе прототипа функции и связывает этот объект с соответствующим параметром `this`. Если вы пренебрегаете использованием префикса `new`, новый объект создан не будет, и параметр `this` окажется связанным с глобальным объектом, а это — серьезная ошибка.

JSLint строго следует соглашению о том, что имена конструкторов должны начинаться с прописной буквы. JSLint не предполагает вызова функции, который может начинаться с прописной буквы, если он не содержит префикс `new`. JSLint не предполагает наличия префикса `new`, используемого с функциями, имена которых не начинаются с прописной буквы.

JSLint не предполагает возможности существования оболочки `new Number`, `new String` или `new Boolean`.

JSLint не ожидает встретить запись `new Object` (вместо этого используйте `{}`).

JSLint не ожидает встретить запись `new Array` (вместо этого используйте `[]`).

Чего JSLint не ищет

JSLint не анализирует поток, чтобы определить, какие значения присваиваются переменным перед их использованием, потому что переменные уже имеют значение (`undefined`), что для многих приложений, в принципе, довольно разумно.

JSLint не производит какого-либо глобального анализа. Верификатор не пытается определить, какие из функций, используемых с `new`, действительно являются конструкторами (за исключением случаев несоблюдения соглашения о прописных буквах).

HTML

Верификатор JSLint способен обрабатывать HTML-текст. Он может проверить JavaScript-код внутри тегов `<script> ...</script>` или обработчика событий. Кроме того, он проверяет HTML-контент и ищет проблемы, которые, как известно, могут повлиять на JavaScript:

- все имена тегов должны быть в нижнем регистре;
- все теги, которым требуется закрывающий тег (например, `</p>`), должны его иметь;
- все теги должны быть правильно вложены;
- литерал `<` должен задаваться символами `&lt;`.

JSLint не так дотошно требует соответствия, как XHTML, хотя делает это строже, чем популярные браузеры.

Кроме того, JSLint отслеживает появление символов `</` в строковых литералах. Вместо этого всегда следует писать `<\/`. Дополнительный слэш игнорируется компилятором JavaScript, но не средством синтаксического разбора HTML-кода. И хотя подобные трюки не должные существовать, он существуют.

Есть опция, позволяющая использовать верхний регистр для имен тегов. Кроме того, есть опция, которая дает возможность использовать встроенный обработчик HTML-событий.

JSON

JSLint может проверить, правильно ли сформированы структуры JSON-данных. Если JSLint обнаруживает, что первым идет символ `{` или `[`, то верификатор строго контролирует правила JSON. Более подробно этот формат описан в приложении Д.

Отчет

Если JSLint в состоянии завершить проверку, генерируется отчет, в котором для каждой функции представлена следующая информация:

- Номер строки, на которой она начинается.
- Ее имя. В случае если это анонимная функция, JSLint постарается «угадать» имя.

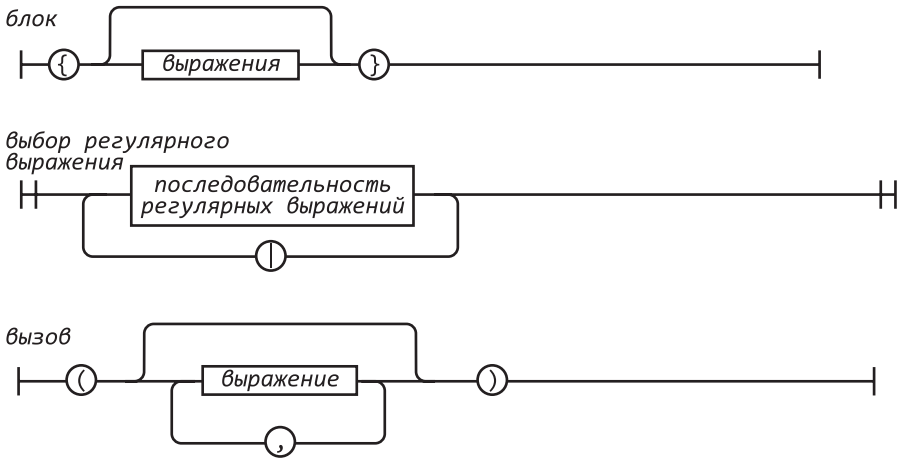
- Параметры.
- Переменные и параметры, объявленные внутри функции и используемые ее внутренними функциями.
- Переменные, объявленные внутри функции и используемые только этой функцией.
- Переменные, объявленные внутри функции, но не используемые. Это может свидетельствовать об ошибке.
- Переменные, используемые функцией, но объявленные в другой функции.
- Глобальные переменные, используемые функцией.
- Метки, используемые функцией.

Кроме того, отчет содержит список имен всех используемых членов.

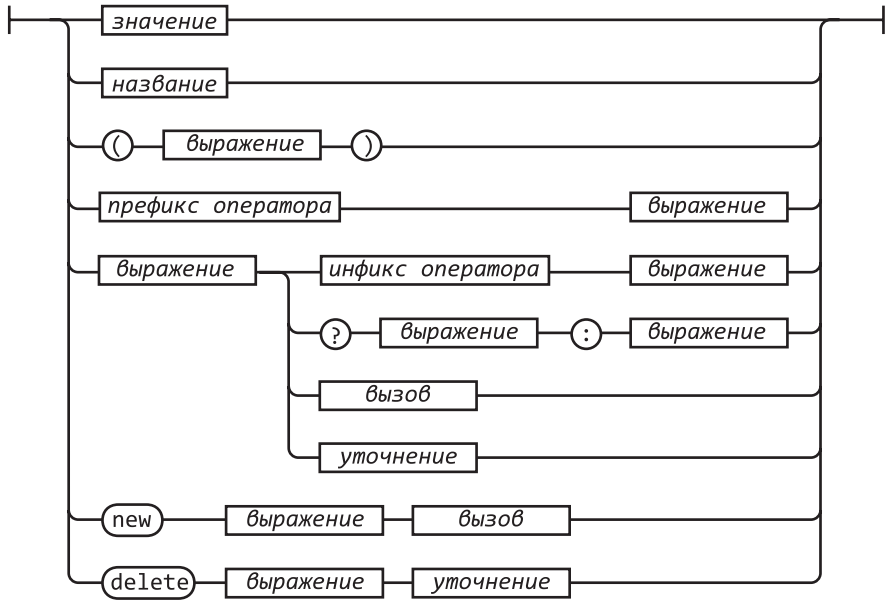
Синтаксические диаграммы

Ты, образ скорби, говоришь без слов!

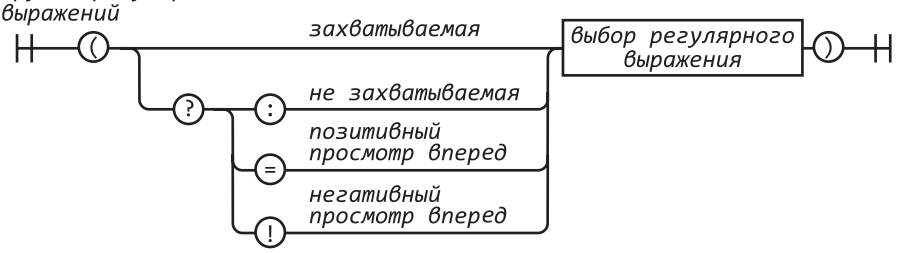
Уильям Шекспир. Тит Андроник



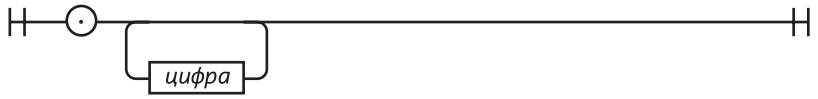
выражение



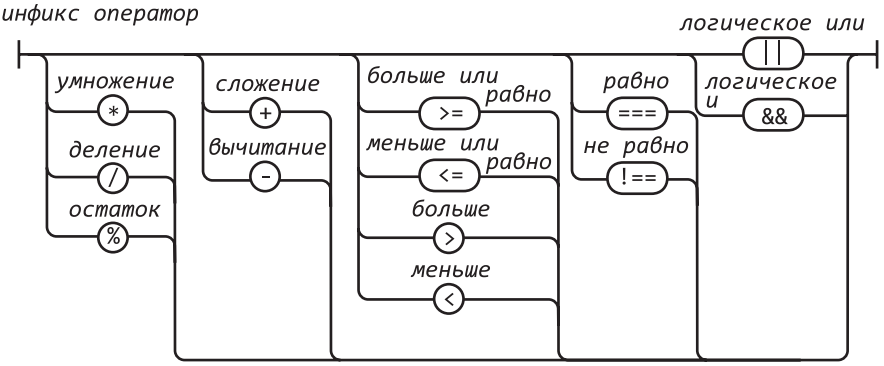
группа регулярных выражений



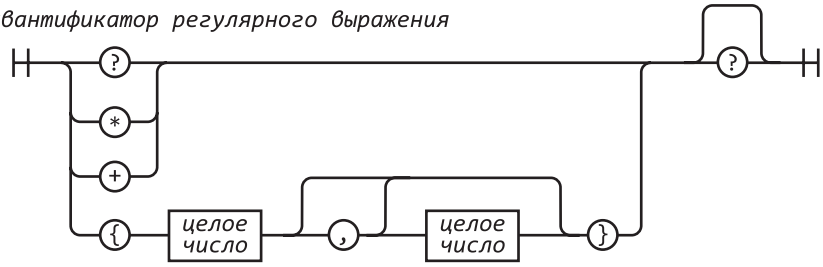
дробь



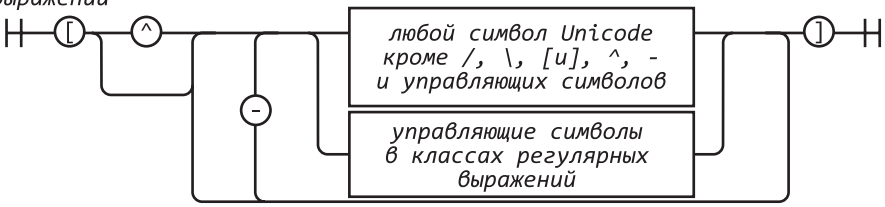
инфикс оператор



квантификатор регулярного выражения

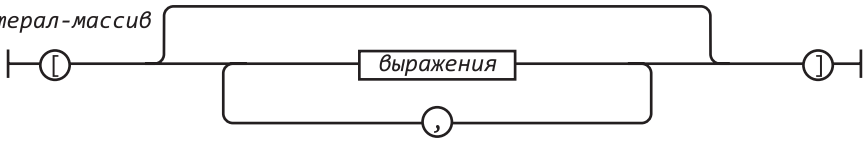


классы регулярных выражений

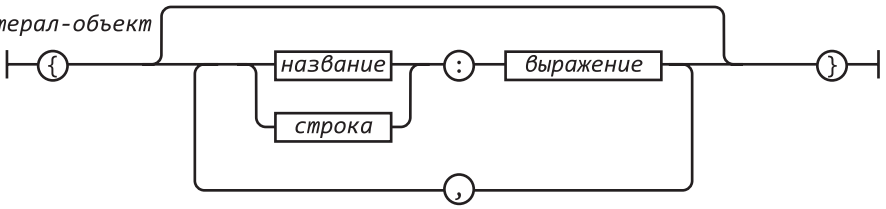


литерал

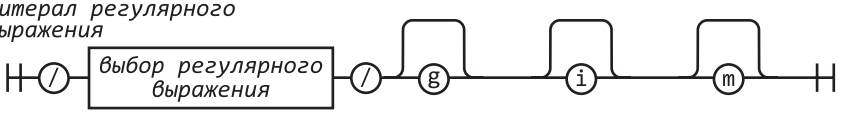
литерал-массив



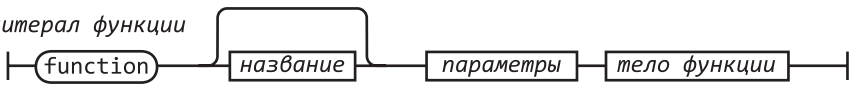
литерал-объект



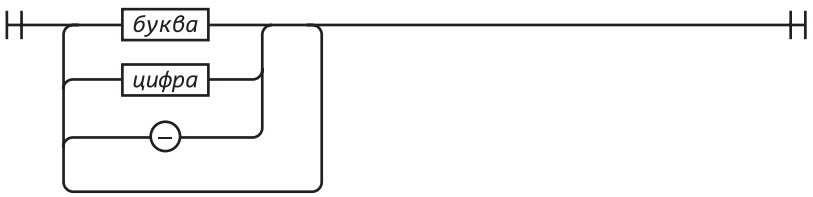
литерал регулярного выражения



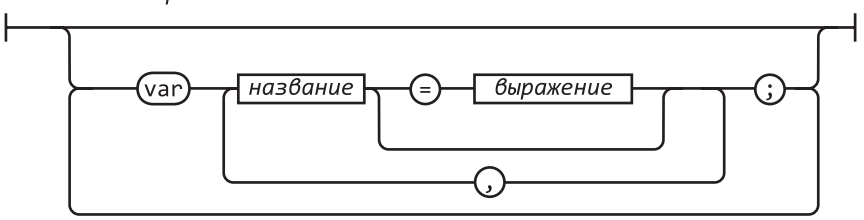
литерал функции



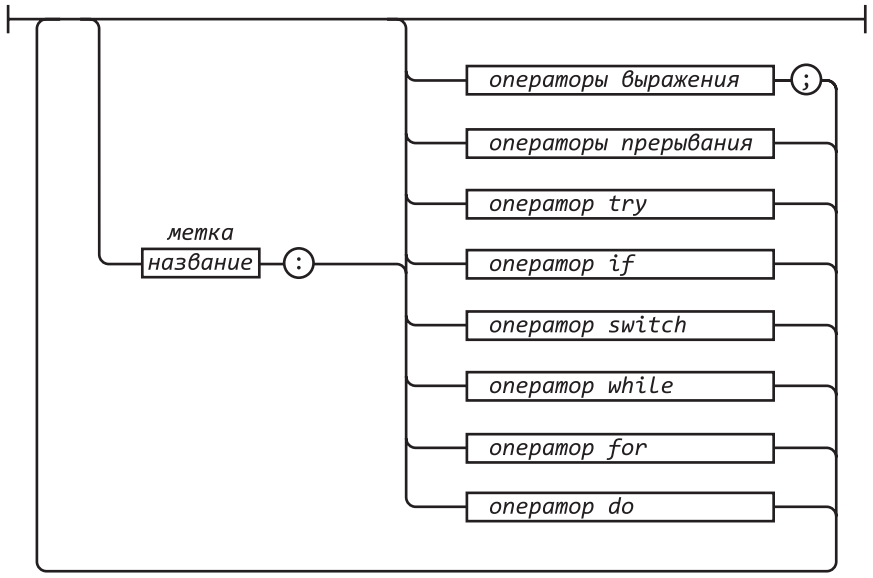
название



объявление переменных

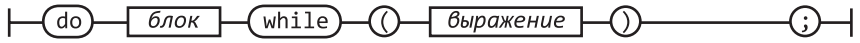


операторы

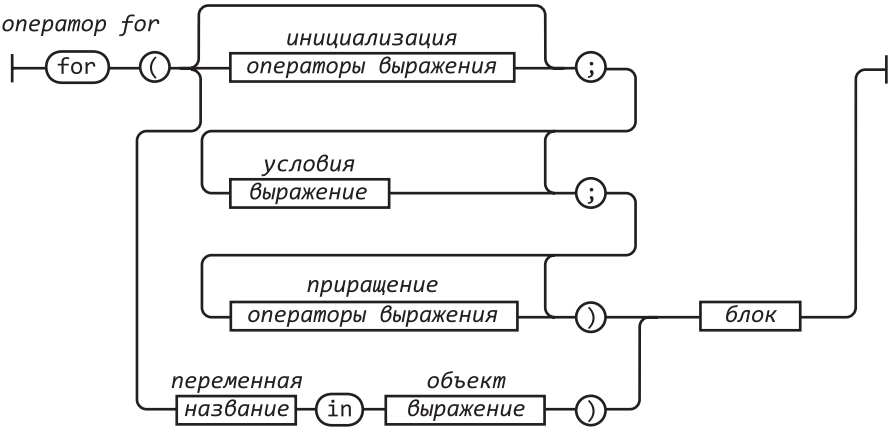


оператор break

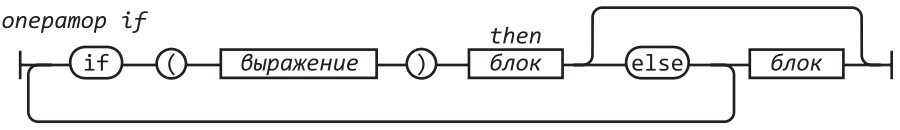
оператор do



оператор for

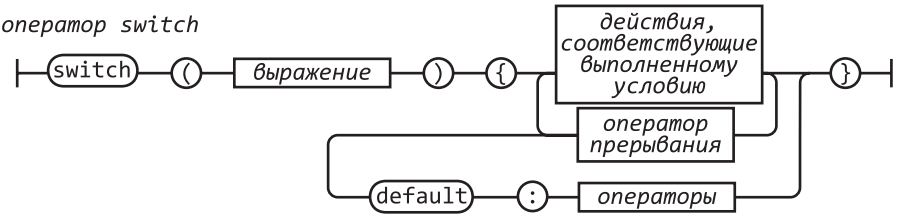


оператор if



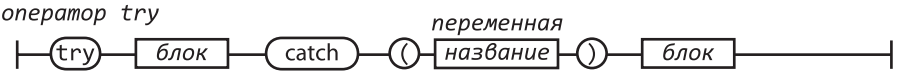
оператор return

оператор switch

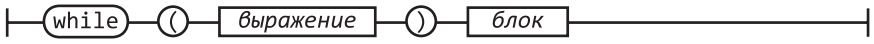


оператор throw

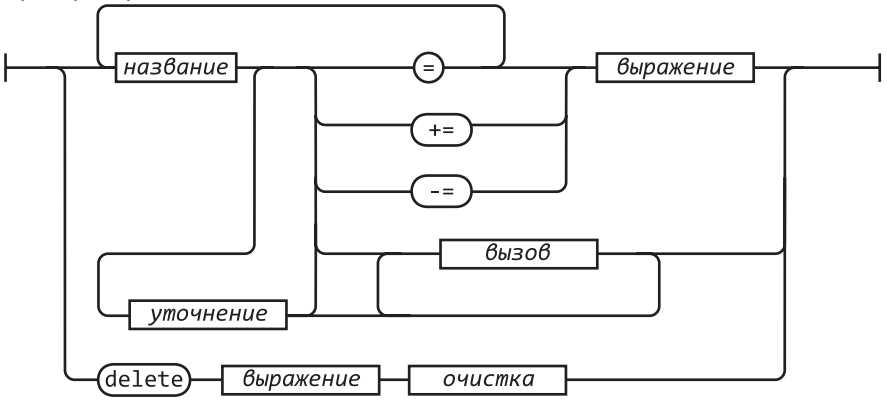
оператор try



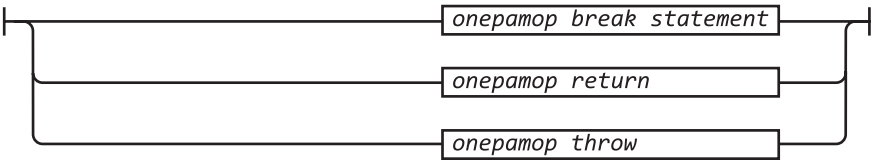
оператор while



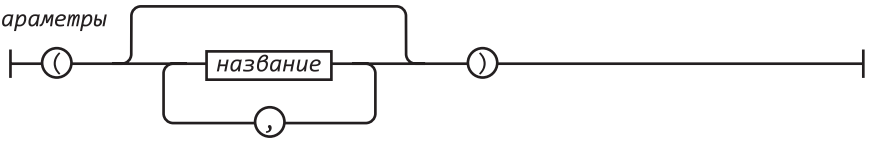
оператор выражения



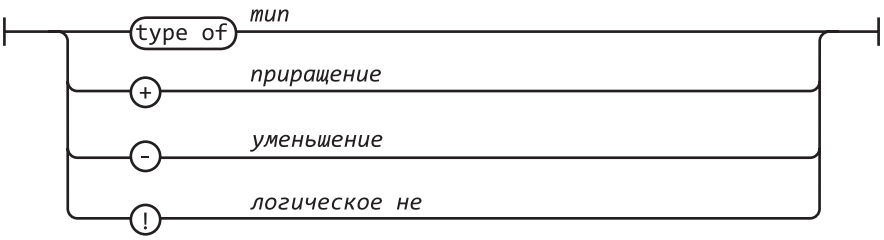
оператор прерывания



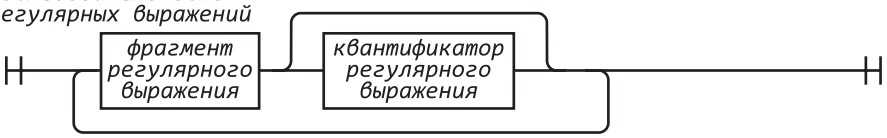
параметры



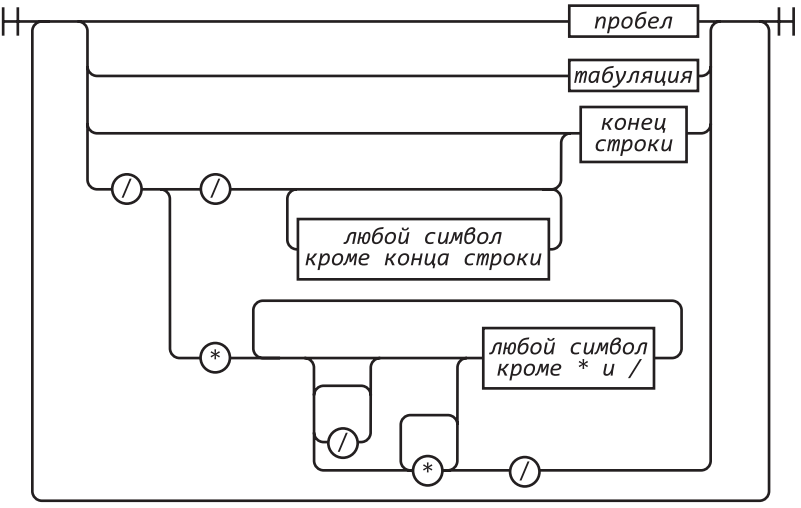
префикс оператор



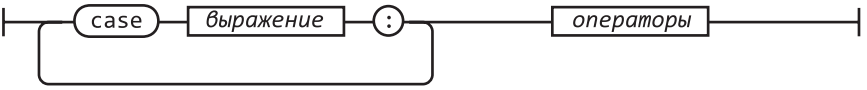
последовательность регулярных выражений



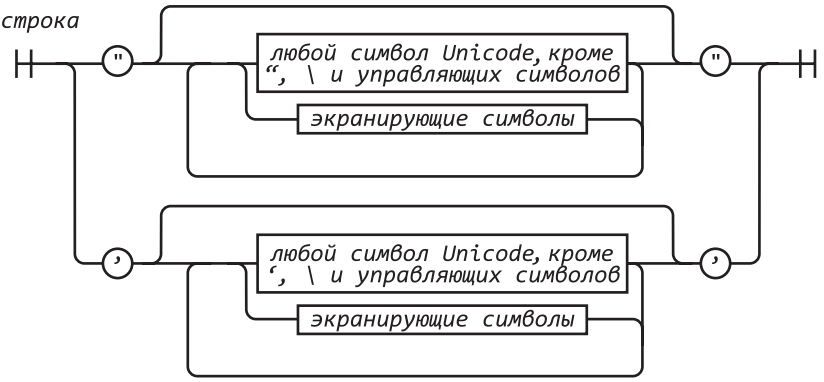
пробел



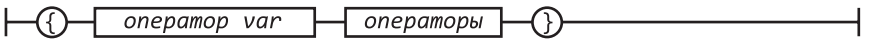
пункт case



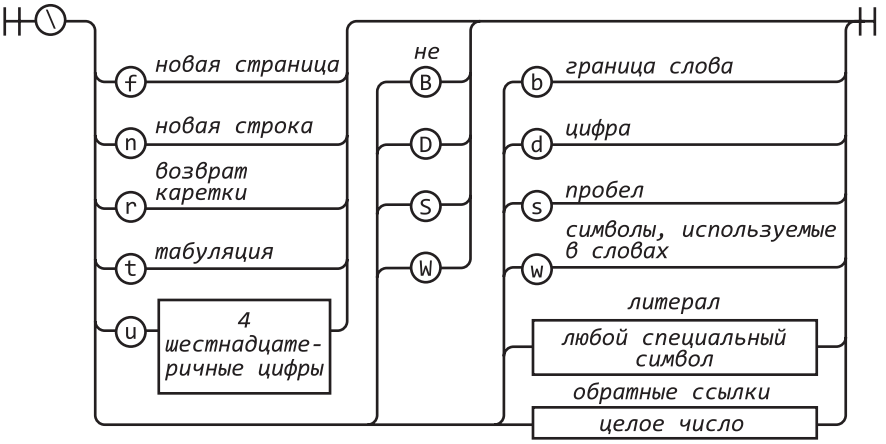
строка



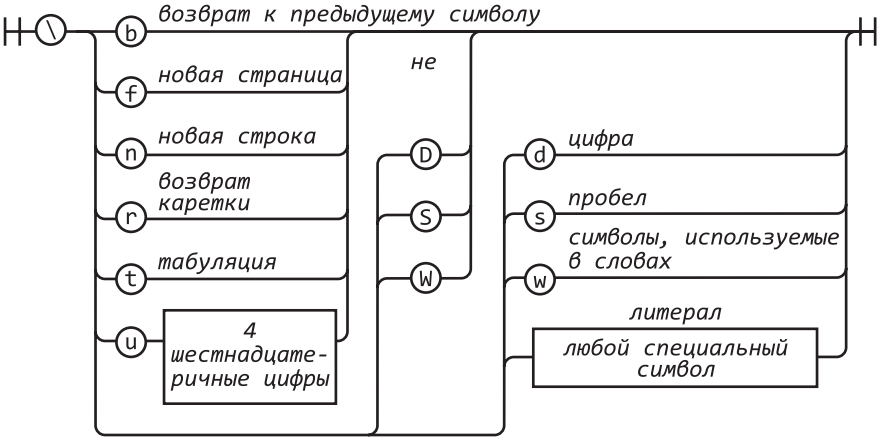
тело функции



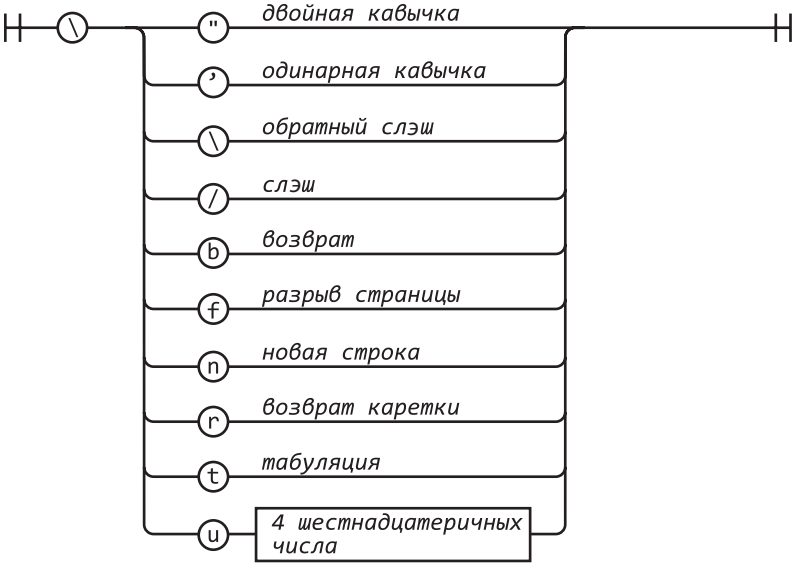
управляющие символы в регулярных выражениях



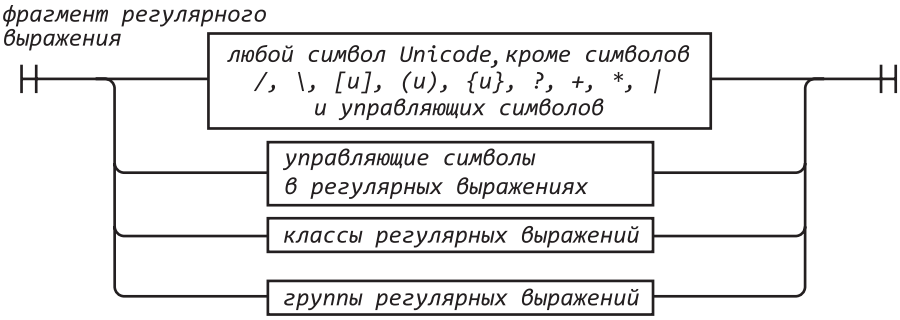
управляющие символы в классах регулярных выражений

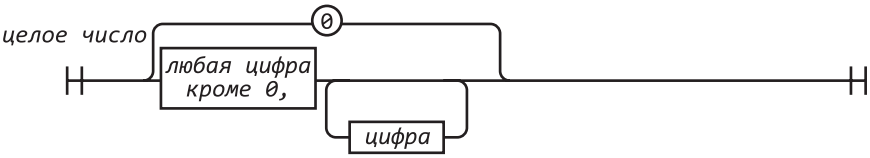


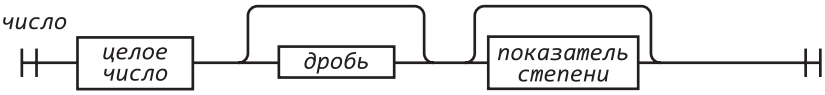
управляющий символ

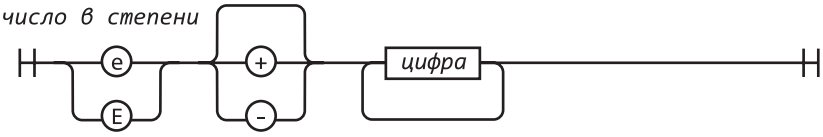


уточнение


фрагмент регулярного выражения
любой символ Unicode, кроме символов /, \, [и], (и), {и}, ?, +, *, | и управляющих символов
управляющие символы в регулярных выражениях
классы регулярных выражений
группы регулярных выражений


целое число
0
любая цифра кроме 0,
цифра


число
целое число
дробь
показатель степени


число в степени
e
E
+
-
цифра

JSON

Прощай. Опасно нам и недосуг
Слова учтивой дружбы говорить,
Обмениваясь нежными речами.
Бог да пошлет нам вольно исполнять
Обряды нежной дружбы!
Прощай, прощай; будь смел и торопись!

Уильям Шекспир. Ричард III

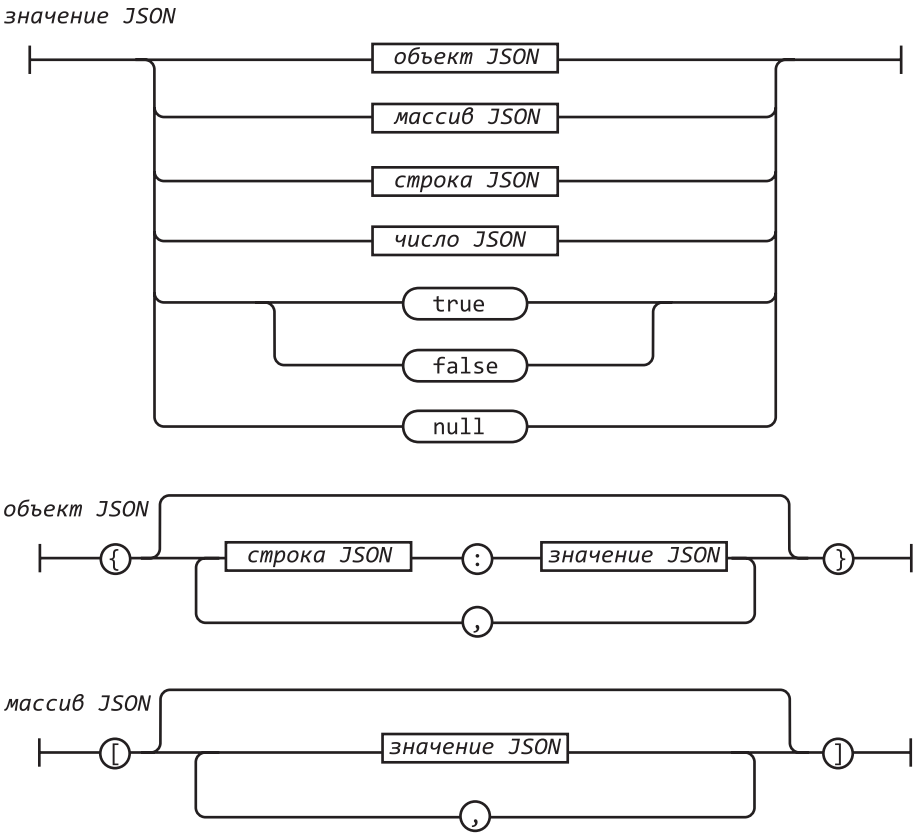
JavaScript Object Notation (JSON) представляет собой простой формат обмена данными, основанный на одной из лучших особенностей JavaScript — литералах объектов. И хотя JSON — это подмножество JavaScript, оно является независимым языком программирования и может использоваться для обмена данными между программами, написанными на любом из современных языков программирования. JSON — это текстовый формат, поэтому он доступен для чтения как людям, так и машинам. Его легко реализовывать и просто использовать. Все материалы, посвященные JSON, можно найти по адресу http://www.JSON.org/.

Синтаксис JSON

JSON имеет шесть видов значений: объекты, массивы, строки, числа, логические значения (`true` и `false`), а также специальное значение `null`. До или после какого-либо значения могут быть пробельные символы (пробел, табуляция, возврат каретки или разрыв строки), это упрощает чтение JSON-текстов людьми. Для снижения затрат на передачу или хранение пробельные символы могут быть опущены.

JSON-объект представляет собой неупорядоченный контейнер пар имя/значение. Имя — это любая строка. Значение — любое JSON-значение, в том числе массив или объект. JSON-объекты могут быть вложенными, но эффективнее оставить их относительно плоским. В большинстве языков есть функции, легко отражающие такие JSON-объекты, как объект, структура, запись, словарь, хэш-таблица, список свойств или ассоциативный массив.

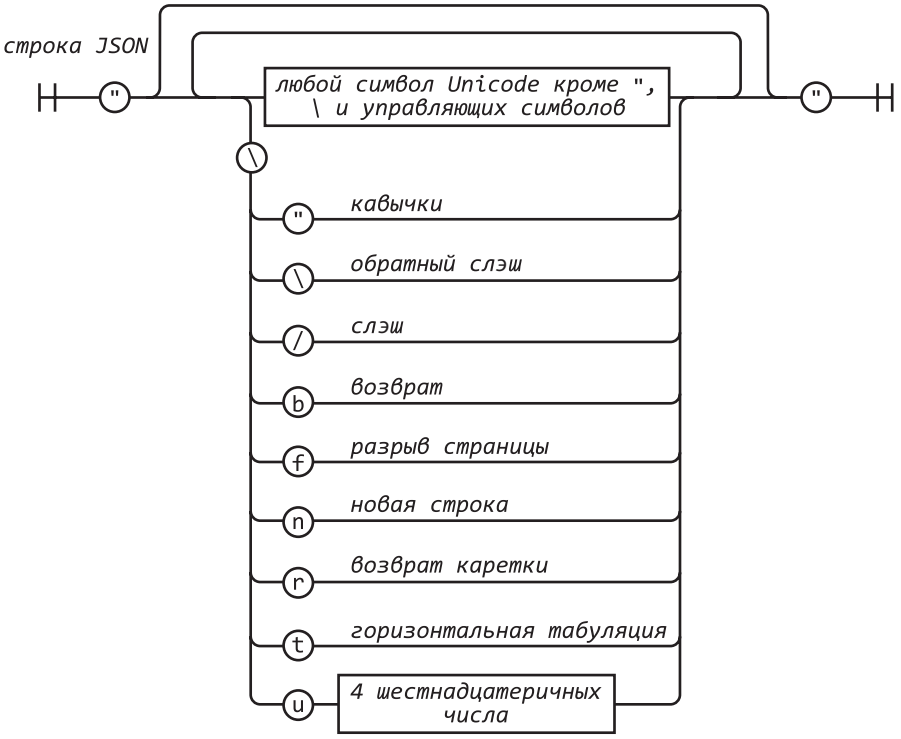
JSON-массив представляет собой упорядоченную последовательность значений. Значение — это любое JSON-значение, в том числе массив или объект. В большинстве языков есть функции, легко отражающие такие JSON-массивы, как массив, вектор, список или последовательность.

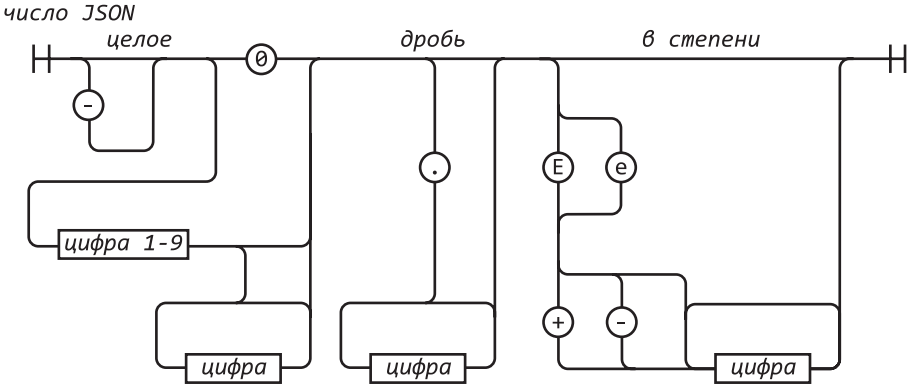


JSON-строка заключается в двойные кавычки, для управления используется символ \. JSON допускает применение символа / для экранирования, поскольку JSON-текст может быть встроен в HTML-тег `<script>`. HTML не позволяет использовать последовательность `</` кроме как в качестве начальных символов тега `</script>`. JSON позволяет использовать символы `<\/`, которые дают тот же результат, но не стоит путать их с HTML-символами.

JSON-числа напоминают числа в JavaScript. Для целых чисел нет значения ноль, так как некоторые языки используют его в восьмеричном счислении. Такая путаница со счислением в формате обмена данными нежелательна. Число может быть целым, вещественным или экспонентой.

Вот и все. Это весь формат JSON. Целью разработки JSON было создать компактное, переносимое, текстовое подмножество JavaScript. Чем меньше всего необходимо для взаимодействия, тем легче взаимодействовать.





```
[
    {
        "first": "Jerome",
        "middle": "Lester",
        "last": "Howard",
        "nick-name": "Curly",
        "born": 1903,              "died": 1952,
        "quote": "nyuk-nyuk-nyuk!"
    },
    {
        "first": "Harry",
        "middle": "Moses",
        "last": "Howard",
        "nick-name": "Moe",
        "born": 1897,
        "died": 1975,
        "quote": "Why, you!"
    },
    {
        "first": "Louis",
        "last": "Feinberg",
        "nick-name": "Larry",
        "born": 1902,
        "died": 1975,
        "quote": "I'm sorry. Moe, it was an accident!"
    }
]
```

Безопасное использование JSON

Поскольку JSON — это JavaScript, этот формат проще всего использовать в веб-приложениях. С помощью функции `eval` JSON-текст может быть превращен в необходимую структуру данных:

```
var myData = eval('(' + myJSONText + ')');
```

(Круглые скобки вокруг JSON-текста помогают обойти неоднозначность в грамматике языка JavaScript.)

Функции `eval` присущи кошмарные проблемы, связанные с безопасностью. Безопасно ли использовать `eval` для синтаксического разбора JSON-текста? Сейчас лучшим способом для передачи данных с сервера в веб-браузер является запрос `XMLHttpRequest`. Однако он позволяет получать данные только с одного сервера, который и создал этот HTML-код. Текст `evaling`, полученный от этого сервера, безопасен не меньше, чем оригинальный HTML-текст. Но предположим, что это вредоносный сервер. Или просто некомпетентный.

Некомпетентный сервер не сможет правильно выполнить JSON-кодирование. Если он создает JSON-текст не с помощью специального JSON-кодера, а просто составляя вместе строки, он может непреднамеренно отправить опасные материалы.

Если он действует как прокси-сервер и просто передает JSON-текст, не определяя, как он сформирован, тоже может послать опасные материалы.

Избежать опасных ситуаций можно, используя метод `JSON.parse` вместо функции `eval` (детали см. по адресу http:// www.JSON.org/json2.js). Если текст содержит опасный фрагмент, `JSON.parse` генерирует исключение. Для защиты от некомпетентности сервера рекомендуется всегда использовать метод `JSON.parse` вместо `eval`. Также это может пригодиться, если браузер обеспечивает безопасный доступ к данным на других серверах.

При взаимодействии между внешними данными и `innerHTML` существует еще одна опасность — это общий Ajax-шаблон сервера, предназначенный для отправки текста HTML-фрагментов, который присваивается свойству `innerHTML` HTML-элементов. Это очень плохо. Если текст содержит HTML-тег `<script>` или его эквивалент, будет исполнен вредоносный сценарий, что опять-таки может быть связано с некомпетентностью сервера.

В чем конкретно состоит опасность? Если вредоносный сценарий сможет запуститься на вашей странице, он получит доступ ко всем состояниям и возможностям страницы. Он сможет взаимодействовать с сервером, а сервер не сможет отличить вредоносные запросы от легитимных. Вредоносный сценарий получит доступ к глобальному объекту, что даст ему доступ ко всем данным приложения, за исключением переменных, скрытых в замыканиях. Он будет иметь доступ к объекту документа, предоставляющему доступ ко всему, что видит пользователь, что даст вредоносному сценарию возможность диалога с пользователем. Адресная строка браузера и вся антифишинговая защита сообщит пользователю, что этому диалогу следует доверять. Объект документа также даст вредоносному сценарию доступ к сети, чтобы загрузить еще более вредоносный сценарий, изучить сайты, находящиеся внутри брандмауэра, или передать полученные секреты на любой сервер мира.

Такая опасная ситуация напрямую зависит от глобального JavaScript-объекта, который является, несомненно, худшим из недостатков JavaScript. Подобная опасность не связана с Ajax, JSON, XMLHttpRequest и Web 2.0 (независимо от того, что из этого используется). Она была в браузере с момента появления JavaScript и будет существовать там, пока язык JavaScript не будет изменен или исправлен. Будьте осторожны.

JSON-парсер

Далее представлена написанная на JavaScript реализация средства синтаксического разбора для JSON (JSON-парсер):

```
var json_parse = function () {

// Эта функция анализирует JSON-текст, возвращая JavaScript-структуру
// данных. Это простое рекурсивное средство синтаксического разбора.
```

продолжение ⇩

```
// Мы определяем функцию внутри другой функции, чтобы избежать
// создания глобальных переменных.

    var at,    // Индекс текущего символа
        ch,    // Текущий символ
        escapee = {
            '\\':     '\\',
            '/':     '/',
            b:        'b',
            f:        '\f',
            n:         '\n',
            r:        '\r',
            t:        '\t'
        },
        text,

        error = function (m) {

// Функция error вызывается, если что-то не так.

            throw {
                name:        'SyntaxError',
                message:     m,
                at:            at,
                text:        text
            };
        },

        next = function (c) {

// Если задан параметр c, необходимо убедиться,
// соответствует ли он текущему символу.

            if (c && c !== ch) {
                error("Expected '" + c + "' instead of '" + ch +     );
            }

// Получаем следующий символ. Если символов больше нет,
// возвращаем пустую строку.

            ch = text.charAt(at);
            at += 1;
            return ch;
        },

        number = function () {

// Анализируем значение number.
```

```
            var number,
                string = '';

            if (ch === '-') {
                string = next('-');
            }
            while (ch >= '0' && ch <= '9') {
                string += ch;
                next();
            }
            if (ch === '.') {
                string +=;
                while (next() && ch >= '0' && ch <= '9') {
                    string += ch;
                }
            }
            if (ch === 'e' || ch === 'E') {
                string += ch;
                next();
                if (ch === '-' || ch === '+') {
                    string += ch;
                    next();
                }
                while (ch >= '0' && ch <= '9') {
                    string += ch;
                    next();
                }
            }
            number = +string;
            if (isNaN(number)) {
                error("Bad number");
            } else {
                return number;
            }
        },

        string = function () {

// Анализируем значение string.

            var hex,
                i,
                string = '',
                uffff;
// Для анализа значения string нужно найти символы " и \.

            if (ch === '"') {
                while (next()) {
```

продолжение ⇩

```
                    if (ch === '"') {
                        next();
                        return string;
                    } else if (ch === '\\') {
                        next();
                        if (ch === 'u') {
                            uffff = 0;
                            for (i = 0; i < 4; i += 1) {
                                hex = parseInt(next(), 16);
                                if (!isFinite(hex)) {
                                    break;
                                }
                                uffff = uffff * 16 + hex;
                            }
                            string += String.fromCharCode(uffff);
                        } else if (typeof escapee[ch] === 'string') {
                            string += escapee[ch];
                        } else {
                            break;
                        }
                    } else {
                        string += ch;
                    }
                }
            }
            error("Bad string");
        },

        white = function () {

// Пропустим пробелы.

            while (ch && ch <= ' ') {
                next();
            }
        },

        word = function () {

// true, false или null.

            switch (ch) {
            case 't':
                next('t');
                next('r');
                next('u');
                next('e');
                return true;
            case 'f':
```

```
                next('f');
                next('a');
                next('l');
                next('s');
                next('e');
                return false;
            case 'n':
                next('n');
                next('u');
                next('l');
                next('l');
                return null;
            }
            error("Unexpected '" + ch + "'");
        },

        value,    // Место хранения функции value.
        array = function () {

// Анализируем значение array.
            var array = [];

            if (ch === '[') {
                next('[');
                white();
                if (ch === ']') {
                    next(']');
                    return array;    // пустой массив
                }
                while (ch) {
                    array.push(value());
                    white();
                    if (ch === ']') {
                        next(']');
                        return array;
                    }
                    next(',');
                    white();
                }
            }
            error("Bad array");
        },

        object = function () {

// Анализируем значение object.

            var key,
```

продолжение ⇩

```
                object = {};

            if (ch === '{') {
                next('{');
                white();
                if (ch === '}') {
                    next('}');
                    return object;    // пустой объект
                }
                while (ch) {
                    key = string();
                    white();
                    next(':');
                    object[key] = value();
                    white();
                    if (ch === '}') {
                        next('}');
                        return object;
                    }
                    next(',');
                    white();
                }
            }
            error("Bad object");
        };

    value = function () {

// Анализируем JSON-значение. Оно не может быть объектом,
// массивом, строкой, числом или словом.

        white();
        switch (ch) {
        case '{':
            return object();
        case '[':
            return array();
        case '"':
            return string();
        case '-':
            return number();
        default:
            return ch >= '0' && ch <= '9' ? number() : word();
        }
    };

// Возвращаем функцию json_parse. Она будет иметь доступ ко всем
// объявленным выше функциям и переменным.
```

```
    return function (source, reviver) {
        var result;

        text = source;
        at = 0;
        ch = ' ';
        result = value();
        white();
        if (ch) {
            error("Syntax error");
        }

// Если используется функция reviver, можно рекурсивно перейти
// к новой структуре, проходя каждую пару имя/значение в функции reviver,
// чтобы, возможно, ее изменить, начиная с временно загруженного
// объекта, который хранит результат в пустом ключе. Если функция
// reviver не используется, просто будет возвращен результат.

        return typeof reviver === 'function' ?
            function walk(holder, key) {
                var k, v, value = holder[key];
                if (value && typeof value === 'object') {
                    for (k in value) {
                        if (Object.hasOwnProperty.call(value, k)) {
                            v = walk(value, k);
                            if (v !== undefined) {
                                value[k] = v;
                            } else {
                                delete value[k];
                            }
                        }
                    }
                }
                return reviver.call(holder, key, value);
            }({'': result}, '') : result;
    };
}();
```

Дуглас Крокфорд

JavaScript: сильные стороны

Перевела с английского А. Лузган

Заведующий редакцией	*А. Кривцов*
Руководитель проекта	*А. Кривцов*
Ведущий редактор	*Ю. Сергиенко*
Литературный редактор	*А. Жданов*
Художественный редактор	*Л. Адуевская*
Корректор	*В. Листова*
Верстка	*Л. Волошина*

ООО «Мир книг», 198206, Санкт-Петербург, Петергофское шоссе, 73, лит. А29.
Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.
Подписано в печать 27.02.12. Формат 70x100/16. Усл. п. л. 14,190. Тираж 2000. Заказ 0000.
Отпечатано по технологии CtP в ОАО «Первая Образцовая типография», обособленное подразделение «Печатный двор».
197110, Санкт-Петербург, Чкаловский пр., 15.

ВАМ НРАВЯТСЯ НАШИ КНИГИ? ЗАРАБАТЫВАЙТЕ ВМЕСТЕ С НАМИ!

У Вас есть свой сайт?
Вы ведете блог?
Регулярно общаетесь на форумах? Интересуетесь литературой, любите рекомендовать хорошие книги и хотели бы стать нашим партнером?
ЭТО ВПОЛНЕ РЕАЛЬНО!

СТАНЬТЕ УЧАСТНИКОМ ПАРТНЕРСКОЙ ПРОГРАММЫ ИЗДАТЕЛЬСТВА «ПИТЕР»!


*Зарегистрируйтесь на нашем сайте в качестве партнера по адресу **www.piter.com/ePartners***


Получите свой персональный уникальный номер партнера


*Выбирайте книги на сайте **www.piter.com,** размещайте информацию о них на своем сайте, в блоге или на форуме и добавляйте в текст ссылки на эти книги (на сайт **www.piter.com**)*

ВНИМАНИЕ! *В каждую ссылку необходимо добавить свой персональный уникальный номер партнера.*

С этого момента получайте 10% от стоимости каждой покупки, которую совершит клиент, придя в интернет-магазин «Питер» по ссылке с Вашим партнерским номером. А если покупатель приобрел не только эту книгу, но и другие издания, Вы получаете дополнительно по 5% от стоимости каждой книги.

Деньги с виртуального счета Вы можете потратить на покупку книг в интернет-магазине издательства «Питер», а также, если сумма будет больше 500 рублей, перевести их на кошелек в системе Яндекс.Деньги или Web.Money.

Пример партнерской ссылки:
http://www.piter.com/book.phtml?978538800282 – обычная ссылка
http://www.piter.com/book.phtml?978538800282**&refer=0000** – партнерская ссылка, где 0000 – это ваш уникальный партнерский номер

Подробно о Партнерской программе ИД «Питер» читайте на сайте WWW.PITER.COM

ИЗДАТЕЛЬСКИЙ ДОМ ПИТЕР®
WWW.PITER.COM

ПРЕДСТАВИТЕЛЬСТВА ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР»
предлагают профессиональную и популярную литературу по различным направлениям: история и публицистика, экономика и финансы, менеджмент и маркетинг, компьютерные технологии, медицина и психология.

РОССИЯ

Санкт-Петербург: м. «Выборгская», Б. Сампсониевский пр., д. 29а
тел./факс: (812) 703-73-73, 703-73-72; e-mail: sales@piter.com

Москва: м. «Электрозаводская», Семеновская наб., д. 2/1, стр. 1
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. Бебеля, д. 11а
тел./факс: (343) 378-98-41, 378-98-42; e-mail: office@ekat.piter.com

Нижний Новгород: тел.: 8 960 187-85-50; e-mail: yashny@yandex.ru

Новосибирск: Комбинатский пер., д. 3
тел./факс: (383) 279-73-92; e-mail: sib@nsk.piter.com

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, 229-68-09; e-mail: pitvolga@smrtlc.ru, pitvolga@mail.ru

УКРАИНА

Киев: Московский пр., д. 6, корп. 1, офис 33
тел./факс: (044) 490-35-69, 490-35-68; e-mail: office@kiev.piter.com

Харьков: ул. Суздальские ряды, д. 12, офис 10
тел./факс: (057) 7584145, +38 067 545-55-64; e-mail: piter@kharkov.piter.com

БЕЛАРУСЬ

Минск: ул. Розы Люксембург, д. 163
тел./факс: (517) 208-80-01, 208-81-25; e-mail: gv@minsk.piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству зарубежных торговых партнеров или посредников, имеющих выход на зарубежный рынок
тел./факс: (812) 703-73-73; e-mail: rodionova.tatyana@piter.com

---

Издательский дом «Питер» приглашает к сотрудничеству авторов
тел./факс издательства: (812) 703-73-72, (495) 974-34-50

Заказ книг для вузов и библиотек
тел./факс: (812) 703-73-73, доб. 6250; e-mail: fokina@piter.com

---

Заказ книг по почте: на сайте www.piter.com; по тел.: (812) 703-73-74, доб. 6225
