Сделайте веб-страницы интерактивными!

3-Е ИЗДАНИЕ

JavaScript

Карманный справочник

O'REILLY®

Дэвид Флэнаган

3-е издание

JavaScript

Карманный справочник

Third Edition

JavaScript

Pocket Reference

David Flanagan

O'REILLY®

Beijing · Cambridge · Farnham - Köln · Sebastopol · Tokyo

3-е издание

JavaScript

Карманный справочник

Дэвид Флэнаган

Москва · Санкт-Петербург · Киев
2013

ББК 32.973.26-018.2.75
Ф73
УДК 681.3.07

Издательский дом "Вильямс"
Главный редактор *С.Н. Тригуб*
Зав. редакцией *В.Р. Гинзбург*
Перевод с английского и редакция канд. техн. наук *А.Г. Сысонюка*

По общим вопросам обращайтесь в Издательский дом "Вильямс"
по адресу: info@williamspublishing.com,
http://www.williamspublishing.com

Флэнаган, Дэвид.

Ф73 JavaScript: карманный справочник, 3-е изд. : Пер. с англ. — М. : ООО "И.Д. Вильямс", 2013. — 320 с. : ил. — Парал. тит. англ.

ISBN 978-5-8459-1830-7 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Дэвид Флэнаган

JavaScript: карманный справочник, 3-е издание

Литературный редактор *Л.Н Красножон*
Верстка *О.В. Мишутина*
Художественный редактор *В.Г. Павлютин*
Корректор *Л.А. Гордиенко*

Подписано в печать 14.12.2012. Формат 70x100/32
Гарнитура Times. Печать офсетная
Усл. печ. л. 12.9. Уч.-изд. л. 10,5
Тираж 1500 экз. Заказ № 653

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9-я линия, 12/28

ООО "И. Д. Вильямс", 127055, г. Москва, ул. Лесная, д. 43, стр. 1

ISBN 978-5-8459-1830-7 (рус.)
ISBN 978-1-449-31685-3 (англ.)

Оглавление

Содержание

Об авторе

Дэвид Флэнаган – JavaScript-программист в компании Mozilla, автор многочисленных книг по языкам программирования, включая JavaScript, Java и Ruby. Получил степень доктора технических наук в Массачусетсском технологическом институте. Ведет блог по адресу `www.davidflanagan.com`.

Изображение на обложке

На обложке книги изображен яванский носорог. Известны пять видов носорогов. Все они имеют крупные размеры, один или два рога, трехпалые конечности и толстую кожу, напоминающую броню. Как и суматранские сородичи, яванские носороги обитают в лесах. Они похожи на индийских носорогов, но немного меньше по размеру и обладают рядом других отличительных признаков.

Введение

JavaScript — это язык программирования для Интернета. Он используется практически на всех современных веб-сайтах. Во всех современных браузерах и клиентских устройствах — настольных компьютерах, игровых консолях, планшетах и смартфонах — имеются интерпретаторы JavaScript, что делает его самым распространенным языком в истории программирования. Он входит в базовую триаду технологий, которые необходимо знать всем разработчикам веб-приложений: HTML (определение содержимого веб-страниц), CSS (определение внешнего вида веб-страниц) и JavaScript (определение поведения веб-страниц). И наконец, в последнее время с появлением технологии Node.js (`http://nodesj.org`) язык JavaScript стал важным средством создания современных веб-серверов.

В карманном справочнике рассматриваются те же темы, что и в более толстых книгах, только в сжатом, лаконичном формате. В главах 1–9 описывается синтаксис языка — типы, значения, переменные, операторы и инструкции, а затем — объекты, массивы, классы и функции. Эти главы будут полезны для всех программистов, как использующих JavaScript на стороне клиента, так и создающих серверные приложения Node.js.

В наши дни любой язык должен иметь стандартную библиотеку функций для выполнения базовых операций, таких как ввод-вывод данных, отображение элементов интерфейса, обработка строк и т.п. В базовой инфраструктуре JavaScript определен минимальный набор библиотек

для работы с текстом, массивами, датами и регулярными выражениями, но средства ввода-вывода данных в него не входят. Функции ввода-вывода (а также другие специальные функции: сетевые, графические и т.п.) обычно предоставляются хостирующей средой, в которую встроен интерпретатор JavaScript. На стороне клиента хостирующей средой чаще всего служит браузер. В главах 1–9 рассматривается минимальный набор встроенных библиотек языка. В главах 10–14 описывается хостирующая среда браузера и обсуждается использование JavaScript на стороне клиента для создания динамических веб-страниц и веб-приложений.

Количество API-функций JavaScript, реализуемых веб-браузерами, за последние несколько лет значительно увеличилось, поэтому рассмотреть их все в книге небольшого объема невозможно. В главах 10–14 описываются наиболее важные и фундаментальные средства клиентской части JavaScript: окна, документы, элементы, стили, события, средства хранения данных и сетевые функции. Освоив их, можно легко перейти к остальным клиентским библиотекам.

Среда программирования Node.js в последнее время становится все более важной и популярной, но в карманном справочнике для нее просто нет места. Ознакомиться с ней можно по адресу `http://msdn.microsoft.com/ru-ru/asp.net/hh548232`. Для справочника по API-функциям в книге карманного формата также нет места. Неплохой справочник доступен здесь:

```
https://developer.mozilla.org/ru/docs/JavaScript
```

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: info@williamspublishing.com
WWW: http://williamspublishing.com

Наши почтовые адреса:

в России: 127055, Москва, ул. Лесная, д. 43, стр.1
в Украине: 03150, Киев, а/я 152

Лексическая структура

Для написания JavaScript-программ используется набор символов Unicode, который включает в себя наборы ASCII и Latin-1 и поддерживается практически всеми языками и платформами.

Язык JavaScript чувствителен к регистру символов. Это означает, что ключевые слова языка, имена переменных и функций и любые другие *идентификаторы* должны вводиться в строго заданном регистре. Например, ключевое слово `while` нельзя ввести как `While` или `WHILE`. Аналогично `online`, `Online`, `OnLine` и `ONLINE` — это четыре разные переменные.

Комментарии

Синтаксис JavaScript поддерживает два вида комментариев. Во-первых, любой текст между символами // и концом строки считается комментарием и игнорируется интерпретатором JavaScript. Во-вторых, любой текст между символами `/*` и `*/` также считается комментарием; комментарии этого вида могут быть многострочными, но не могут быть вложенными. Следующие строки кода являются допустимыми комментариями JavaScript.

```
// Это однострочный комментарий
/* Комментарий второго вида */ //И еще один...
/*
```

```
 * Это еще один комментарий второго вида,
 * расположенный в нескольких строках.
 */
```

Идентификаторы и зарезервированные слова

Идентификатор — это имя чего-либо. В JavaScript идентификаторы используются для именования переменных, функций, меток и других сущностей. Идентификатор в коде JavaScript может начинаться с буквы, символа подчеркивания (_) или символа доллара ($). Начиная со второго символа можно использовать буквы, цифры, символы подчеркивания и доллара.

Зарезервированные слова — это идентификаторы, служащие ключевыми словами языка. Их нельзя использовать в программе в качестве идентификаторов других сущностей. Ниже перечислены идентификаторы JavaScript.

```
break
case
catch
continue
debugger
default
delete
do
else
false
finally
for
function
if
in
instanceof
new
```

```
null
return
switch
this
throw
true
try
typeof
var
void
while
with
```

Кроме того, зарезервировано несколько ключевых слов, которые сейчас не используются, но могут понадобиться в будущих версиях. В спецификации ECMAScript 5 зарезервированы следующие дополнительные слова.

```
class
const
enum
export
extends
import
super
```

Строгий режим (strict mode) налагает некоторые ограничения на использование приведенных ниже идентификаторов. Они не являются зарезервированными словами, но их нельзя использовать в качестве имен переменных, функций и параметров.

```
arguments
eval
```

В спецификации ECMAScript 3 зарезервированы все ключевые слова Java. Это ограничение отменено в ECMAScript 5, но, если вы планируете выполнять код

в реализациях ECMAScript 3, следующие ключевые слова использовать не рекомендуется.

```
abstract
boolean
byte
char
class
const
double
enum
export
extends
final
float
goto
implements
import
int
interface
long
native
package
private
protected
public
short
static
super
synchronized
throws
transient
volatile
```

Необязательные точки с запятой

Как и во многих других языках программирования, в JavaScript для разделения инструкций (см. главу 4) используется точка с запятой (;). Смысл кода должен быть

очевидным: без разделителя (точки с запятой) конец одной инструкции может выглядеть как начало другой, и наоборот. В JavaScript обычно можно опустить точку с запятой между двумя инструкциями, если они находятся в разных строках. Кроме того, точку с запятой можно опустить в конце программы и перед закрывающей фигурной скобкой }. Однако многие программисты предпочитают всегда ставить точку с запятой (так делается и в данной книге), чтобы явно обозначить конец инструкции, даже если точка с запятой в этом месте необязательна. Другой стиль программирования предполагает опускание точек с запятыми везде, где это возможно, и добавление их только в тех немногих местах, где они обязательны. Выбор стиля зависит от ваших предпочтений, но есть несколько факторов, о которых необходимо помнить.

Рассмотрим приведенный ниже код. Две инструкции находятся в разных строках, поэтому первая точка с запятой необязательна.

```
a = 3;
b = 4;
```

Однако в следующем коде первая точка с запятой обязательна.

```
a = 3; b = 4;
```

Обратите внимание на то, что интерпретатор JavaScript не считает каждый разрыв строки точкой с запятой. Обычно он интерпретирует разрыв строки как точку с запятой, только если не может выделить следующую лексему и продолжить анализ кода. В частности, разрыв строки интерпретируется как точка с запятой после ключевых слов `return`, `break` и `continue`, перед операторами `++` и `--`, а также если следующий непробельный сим-

вол невозможно интерпретировать как продолжение текущей инструкции.

Описанные правила окончания инструкций приводят к ряду неожиданных результатов. Приведенный ниже код выглядит, как две отдельные инструкции, разделенные разрывом строки.

```
var y = x + f
(a+b).toString()
```

Однако скобки во второй строке можно интерпретировать как аргументы вызова функции `f`, расположенной в первой строке, поэтому JavaScript интерпретирует код следующим образом.

```
var y = f(a+b).toString();
```

Типы данных, значения и переменные

Фактически вся работа компьютерных программ заключается в манипулировании *значениями*, такими как число `3.14` или строка `Hello World`. Виды или категории значений, которыми можно манипулировать в программе, называются *типами*. Когда программе нужно сохранить значение для использования в будущем, она присваивает его *переменной* (часто пишут также, что значение *сохраняется в переменной*). С каждой переменной ассоциировано символическое имя (идентификатор), позволяющее в других местах программы ссылаться на значение, хранящееся в данной переменной.

Типы JavaScript делятся на две категории: *примитивные* и *объектные*. К примитивным относятся числовой тип, строка текста (строковый тип) и булев тип, который может принимать значения `true` и `false`. В главах 5–7 рассматриваются три вида объектных типов.

В отличие от строго типизированных языков, таких как Java или C#, интерпретатор JavaScript довольно свободно преобразует типы значений. Например, если программа ожидает строку, но получает число, она автоматически преобразует число в строку. Правила преобразования типов рассматриваются далее.

Переменные JavaScript *нетипизированные*. Это означает, что переменной можно присвоить значение любого типа,

причем позже этой же переменной можно будет присвоить значение другого типа. Переменная *объявляется* с ключевым словом `var`. В программе на JavaScript используется *лексическая область видимости*. Это означает, что переменная, объявленная вне функции, является *глобальной* и видна в любой точке программы. Переменная, объявленная внутри функции, видна только для кода функции. Более подробно объявление переменных рассматривается далее.

Числа

В отличие от многих других языков программирования, в JavaScript не различаются целочисленные значения и значения с плавающей точкой[1]. В программе JavaScript все числа представлены 64-разрядными значениями с плавающей точкой (стандарт IEEE 754) в диапазоне значений ±1.7976931348623157E+308. Согласно этому же стандарту наименьшее число равно ±5E-324.

Числовой формат JavaScript позволяет точно представить все целые значения в диапазоне от –9007199254740992 (-2^{53}) до 9007199254740992 (2^{53}) включительно. Можно использовать также целочисленные значения вне этого диапазона, но будет потеряна точность в последних разрядах. Однако учтите, что определенные операции (такие, как индексирование массивов и побитовые операции; см. главу 3) выполняются с 32-разрядными целыми числами.

Если число записано непосредственно в коде, оно называется *числовым литералом*. Синтаксис JavaScript поддерживает числовые литералы в нескольких форматах.

[1] В программе на JavaScript десятичным разделителем всегда служит точка, даже если в локализованной версии операционной системы в качестве десятичного разделителя определен другой символ (например, запятая в русифицированной системе). — *Примеч. ред.*

Если перед числовым литералом стоит знак "минус", значит, число отрицательное.

В программе JavaScript десятичное целое число записывается как последовательность цифр, например так.

```
0
1024
```

Кроме десятеричных, JavaScript распознает шестнадцатеричные значения. Шестнадцатеричный литерал начинается с префикса `0x` или `0X`, после которого расположена строка шестнадцатеричных цифр. Набор шестнадцатеричных цифр включает цифры от `0` до `9` и буквы от `a` (или `A`) до `f` (или `F`). Ниже приведены примеры шестнадцатеричных литералов.

```
0xff // в десятичной системе 15*16+15=255
0xCAFE911
```

Литералы с плавающей точкой могут содержать точку. В них применяется традиционный синтаксис действительных чисел. Действительное значение представлено целой и дробной частями, разделенными десятичной точкой.

Литералы с плавающей точкой можно записывать в экспоненциальной нотации: после действительного литерала (он называется *мантиссой*) расположена буква `e` (или `E`), затем — необязательный знак "минус" и целое число (оно называется *порядком*), обозначающее степень десяти. Число равно мантиссе, умноженной на 10 в степени порядка.

Более подробный синтаксис числа в экспоненциальной форме приведен ниже.

```
[цифры][.цифры][(E|e)[(+|-)]цифры]
```

Вот примеры числовых литералов с плавающей точкой.

```
3.14
6.02e23 // =6.02×10^23
1.4738223E-32 // =1473822×10^-32
```

Программы JavaScript работают с числами с помощью математических операторов, предоставляемых языком. К ним относятся сложение (+), вычитание (-), умножение (*), деление (/) и деление по модулю (%, этот оператор возвращает остаток целочисленного деления). Более подробно эти операторы рассматриваются в главе 3.

Кроме базовых арифметических операторов, JavaScript поддерживает более сложные математические операции, предоставляемые посредством набора функций и констант, определенных как свойства объекта `Math`.

```
Math.pow(2,53)    // => 9007199254740992, степень
Math.round(.6)    // => 1.0, округление до ближайшего
Math.ceil(.6)     // => 1.0, округление вверх
Math.floor(.6)    // => 0.0, округление вниз
Math.abs(-5)      // => 5, абсолютное значение
Math.max(x,y,z)   // максимальный аргумент
Math.min(x,y,z)   // минимальный аргумент
Math.random()     // псевдослучайное число 0<=x<1.0
Math.PI           // число p
Math.E            // число e
Math.sqrt(3)      // квадратный корень
Math.pow(3,1/3)   // возведение в дробную степень
Math.sin(0)       // синус
Math.log(10)      // натуральный логарифм десяти
Math.log(100)/Math.LN10 // логарифм по основанию 10
Math.log(512)/Math.LN2  // логарифм по основанию 2
Math.exp(3)             // экспонента
```

В случае переполнения, потери значимости или деления на нуль арифметические операторы JavaScript не генерируют ошибку. Если результат операции больше, чем самое большое допустимое число (т.е. при переполнении), оператор возвращает специальное значение `Infinity`, интерпретируемое как бесконечность. Если же это число отрицательное, оператор возвращает `-Infinity`. Бесконечные значения обрабатываются так же, как и в

математике: сложение, вычитание, умножение или деление таких значений приводит, опять же, к бесконечности (но может измениться знак перед числом).

Деление на нуль не считается ошибкой в JavaScript. Оператор всего лишь возвращает положительную или отрицательную бесконечность. Но из этого правила есть одно важное исключение: при делении нуля на нуль результатом не может быть какое-либо определенное значение, поэтому оператор возвращает специальное значение `NaN` (от англ. "not a number" – не число), считающееся неопределенным. Кроме того, значение `NaN` возвращается при попытке деления бесконечности на бесконечность, при вычислении квадратного корня отрицательного числа или при передаче арифметическому оператору нечислового операнда, который невозможно преобразовать в число.

В JavaScript глобальные предопределенные переменные `Infinity` и `NaN` интерпретируются как положительная бесконечность и нечисловое значение.

В JavaScript значение `NaN` обладает одной необычной особенностью: оно не равно никакому другому значению, включая себя же. Следовательно, чтобы выяснить, содержит ли переменная `x` значение `NaN`, нельзя записать `x==NaN`. Правильная запись выглядит так: `x!=x`. Данное выражение равно `true`, только если `x` равно `NaN`. Можно также применить функцию `isNaN()`, которая возвращает `true`, если аргумент равен `NaN` или является нечисловым значением, т.е. строкой (или объектом), которую нельзя преобразовать в число. Аналогичная функция `isFinite()` возвращает `true`, если аргумент является числом, отличным от `NaN`, `Infinity` или `-Infinity`.

В математике количество действительных чисел бесконечно, но в плавающем формате JavaScript количество

чисел, имеющих точное представление, ограничено (их количество равно 18437736874454810627). Это означает, что действительные числа в JavaScript не являются тем же, что действительные числа в математике; они лишь представляют их с определенной точностью. Впрочем, точности представления вполне достаточно для любой практической задачи — не существует ни одной физической величины, которую можно было бы измерить с такой точностью.

Текст

Строка — это неизменяемая упорядоченная последовательность 16-битовых значений, каждое из которых обычно представляет символ Unicode. Строки — это тип JavaScript, представляющий текст. *Длина строки* — это количество хранящихся в ней 16-битовых значений (они называются *элементами строки*). В строках (и массивах строк) JavaScript используется индексация с нуля, т.е. первое значение находится в позиции `0`, второе — в позиции `1` и т.д. *Пустая строка* — это строка с нулевой длиной (не путайте ее со строкой, содержащей только пробелы). В JavaScript нет специального типа, представляющего один элемент строки. Для представления одного 16-битового значения используйте строку единичной длины.

Строковые литералы

Чтобы записать строковый литерал, достаточно заключить любую последовательность символов в одинарные (') или двойные (") кавычки. В строке, ограниченной парой одинарных кавычек, можно использовать двойные кавычки, и наоборот: одинарные кавычки можно использовать в строке, заключенной в пару двойных кавычек. Ниже приведены примеры строковых литералов.

```
  // Это пустая строка
'mane="myform"'
"Вам нравятся книги O'Reilly?"
"Эта строка\nсостоит из двух строк на экране"
"n = 3.14"
```

Обратная косая черта (\) в строковых литералах имеет специальное назначение: вместе с символом, расположенным после нее, она представляет символ, который нельзя представить в строковом литерале иным образом. Такие последовательности называются *специальными*, или *Esc-последовательностями*. Например, специальная последовательность \n представляет конец текущей строки и вставку новой.

Еще один пример: последовательность \' представляет одинарную кавычку. Эта последовательность полезна, когда нужно добавить одинарную кавычку в строковый литерал, заключенный в пару одинарных кавычек.

В табл. 2.1 перечислены все специальные последовательности JavaScript. Два последних специальных символа позволяют представить как шестнадцатеричное число любой символ, определенный в наборах Latin-1 и Unicode. Например, последовательность \xA9 представляет символ авторского права (©), который в наборе Latin-1 имеет шестнадцатеричный код A9. Аналогично последовательность \u представляет произвольный символ Unicode, определенный четырьмя шестнадцатеричными цифрами. Например, последовательность \u03c0 представляет символ π.

Таблица 2.1. Специальные последовательности JavaScript

Последовательность	Представляемый символ
\0	Символ NUL (\u0000); не путайте с null
\b	Клавиша возврата (Backspace, \u0008)

Окончание табл. 2.1

Последовательность	Представляемый символ
\t	Горизонтальная табуляция (\u0009)
\n	Новая строка (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Одинарная кавычка (\u0027)
\\	Обратная косая черта (\u005C)
\xXX	Символ Latin-1, представленный двумя шестнадцатеричными цифрами
\uXXX	Символ Unicode, представленный четырьмя шестнадцатеричными цифрами

Если обратная косая черта находится перед любым символом, не указанным в табл. 2.1, она игнорируется (конечно, в будущих версиях JavaScript могут появиться новые специальные последовательности). Например, последовательность \# интерпретируется как #. Спецификация ECMAScript 5 позволяет вставлять обратную косую черту перед разрывом строки для размещения строкового литерала в нескольких строках кода.

В JavaScript встроены средства объединения строк. Если записать оператор + с числами, он вычислит их сумму. Если же записать его со строками, он объединит их в одну строку. Эта операция называется *конкатенацией*. Ниже приведен пример конкатенации строк два и слова.

```
msg="два"+"слова"; // => "два слова"
```

Для выяснения длины строки (количества хранящихся в ней 16-битовых значений) используется свойство `length` объекта строки. Например, для вычисления длины строки `s` нужно написать следующее.

```
s.lenhth
```

Кроме свойства `length`, объект строки имеет ряд методов, примеры которых приведены ниже.

```
var s = "слова для теста" // создание строки
s.charAt(0)           // => "с": первый символ
s.charAt(s.length-1) // => "а": последний символ
s.substring(1,4)      // => "лов": символы 2, 3 и 4
s.slice(1,4)          // => "лов": то же
s.slice(-3)           // => "ста": последние три символа
s.indexOf("l")        // => 1: позиция символа л
s.lastIndexOf("a")    // => 1: позиция последнего а
s.indexOf("a", 3)     // => 3: позиция а после 3-й
s.split(", ")         // => "слова", "для", "теста"
s.replace("с", "С")   // => "Слова для теСта":
                      // замена вхождений
s.toUpperCase()       // => "СЛОВА ДЛЯ ТЕСТА"
```

Не забывайте, что в JavaScript строки неизменяемые. Методы `replace()` и `toUpperCase()` создают новые строки, а не изменяют существующие.

В ECMAScript 5 строки можно интерпретировать как массивы, доступные только для чтения. Программа может считывать отдельные символы (16-битовые значения) из строки с помощью квадратных скобок (вместо метода `charAt()`).

```
s = "два слова";
s[0]           // => "д"
s[s.length-1]  // => "а"
```

Булевы значения

Булево значение представляет одно из двух состояний: "истина" или "ложь", "включено" или "выключено", "да" или "нет" и т.п. В JavaScript определены два булевых значения: `true` и `false`.

Обычно булево значение генерируется в результате сравнения двух сущностей в программе, например проверки на равенство.

```
a == 4
```

Этот код проверяет, равно ли значение, хранящееся в переменной `a`, числу `4`. Если равно, результатом сравнения будет значение `true`. В противном случае оператор сравнения возвращает `false`.

В коде JavaScript булевы значения обычно используются в управляющих конструкциях. Например, инструкция `if/else` выполняет одно из двух действий в зависимости от того, чему равно булево значение — `true` или `false`. Обычно оператор, вычисляющий булево значение, включен непосредственно в инструкцию, в которой оно используется.

```
if (a == 4)
  b = b + 1;
else
  a = a + 1;
```

Этот код проверяет, равно ли `a` четырем. Если равно, то код увеличивает на единицу переменную `b`, а если нет, то переменную `a`.

Далее будет показано, что любое значение JavaScript может быть автоматически (неявно) преобразовано в булево. Приведенные ниже значения разных типов преобразуются в `false`.

```
undefined
null
0
-0
NaN
""   // пустая строка
```

Все другие значения, включая все объекты и массивы, неявно преобразуются в `true`. Каждый раз, когда интерпретатор JavaScript ожидает булево значение, он преобразует полученное значение любого типа в `false` или `true` согласно приведенному выше правилу.

Предположим, переменная `o` содержит либо объект, либо значение `null`. Проверить, какое из этих двух значений она содержит, можно с помощью следующего оператора сравнения.

```
if (o !== null)...
```

Оператор проверки на неравенство возвращает `true` или `false`. Однако можно опустить оператор и полагаться на тот факт, что `null` неявно преобразуется в `false`, а любой объект — в `true`.

```
if (o)...
```

В первом случае тело инструкции `if` вычисляется, только если переменная `o` не равна `null`. Во втором случае условие менее жесткое: тело инструкции вычисляется, если переменная `o` не равна `false` или любому другому значению, которое преобразуется в `false`, такому как `null` или `undefined`. Какую из этих двух инструкций лучше применить в конкретном случае, зависит от того, какие значения присваиваются переменной `o`. Если код должен отличать `null` от нуля или пустой строки, необходимо применить явную проверку на неравенство (первую инструкцию).

Значения null и undefined

Ключевое слово null имеет особую интерпретацию: оно обычно сообщает об отсутствии какого-либо значения. Оператор typeof, получив значение null, возвращает строку object, которая "намекает" на то, что null интерпретируется как специальный объект с названием "нет объекта". Однако на практике null обычно считается не объектом, а единственным значением собственного типа, который можно использовать для указания на отсутствие определенного значения любого типа: числа, строки или объекта. Ключевое слово null есть практически в любом языке программирования, но в некоторых языках оно может выглядеть немного иначе, например nil или Null.

В JavaScript есть еще одно специальное значение, undefined, указывающее на отсутствие значения, но в более глубоком смысле. Это значение имеет переменная, которая не была инициализирована. Это также значение, возвращаемое в результате запроса несуществующего свойства объекта или элемента массива. Кроме того, значение undefined возвращается функцией, в теле которой не определено возвращаемое значение. Это же значение передается функции в качестве параметра, если при вызове аргумент не был предоставлен. Фактически идентификатор undefined обозначает предопределенную глобальную переменную (а не ключевое слово языка, как null), инициализированную неопределенным значением. Оператор typeof, получив значение undefined, возвращает undefined. Это говорит о том, что значение undefined является единственным членом специального типа.

Несмотря на все эти отличия, оба ключевых слова — null и undefined — указывают на отсутствие определенного значения и часто взаимозаменяемы. Оператор проверки на равенство (==) считает их равными (для их

различения используйте оператор проверки на строгое равенство ===). Оба этих значения преобразуются в булев тип false в логических выражениях. Ни null, ни undefined не имеет ни свойств, ни методов. Применение точки или квадратных скобок для обращения к свойству или методу этих значений приводит к генерации ошибки TypeError.

Глобальный объект

В предыдущем разделе были рассмотрены примитивные типы и значения JavaScript. Объектным типам — объектам, массивам и функциям — посвящено несколько следующих глав. Но есть один важный объектный тип, который нужно рассмотреть уже сейчас. *Глобальный объект* — это встроенный объект JavaScript, предназначенный для очень важной цели: свойства этого объекта являются глобальными сущностями, доступными в любом месте JavaScript-программы. Когда интерпретатор JavaScript начинает обрабатывать код (при загрузке браузером новой страницы), он создает глобальный объект и присваивает ему набор следующих сущностей:

- глобальные свойства, такие как undefined, Infinity и NaN;
- глобальные функции, такие как isNaN(), parseInt() и eval();
- конструкторы, такие как Date(), RegExp(), String(), Object() и Array();
- предопределенные объекты, такие как Math и JSON (часто их также называют глобальными).

Начальные свойства глобального объекта не являются зарезервированными ключевыми словами, но к ним сле-

дует относиться как к таковым. В данной главе некоторые из таких свойств уже упоминались. Большинство других глобальных свойств рассматриваются в соответствующих главах.

В коде верхнего уровня (т.е. не в теле функции) можно использовать ключевое слово `this` для ссылки на глобальный объект.

```
var global = this;
```

В коде JavaScript на стороне клиента объект `Window` является глобальным. Он содержит свойство `window`, которое указывает на свой же объект `Window` и может быть использовано для ссылки на глобальный объект. В объекте `Window` определены базовые глобальные свойства и ряд свойств, специфичных для браузера и клиентского кода JavaScript (см. главу 10).

При первом создании глобального объекта создаются все предопределенные глобальные значения JavaScript. Кроме того, этот объект содержит глобальные свойства, определенные в программе. Если в коде JavaScript объявлены глобальные переменные, они являются свойствами глобального объекта.

Преобразование типов

Интерпретатор JavaScript весьма гибок по отношению к типам значений. Вы уже видели это на примере булевых значений: когда код ожидает булево значение, он может получить значение любого типа, а интерпретатор преобразует его в булев тип. Некоторые значения преобразуются в `true`, а некоторые — в `false`. То же самое справедливо для всех других типов. Например, когда код ожидает строку, интерпретатор преобразует в строку полученное значение любого типа. Если код ожидает число, интер-

претатор пытается преобразовать полученное значение в число. Если он не может сделать это, код получит значение NaN. Рассмотрим ряд примеров.

```
10 +   объектов" // => "10 объектов"
                 // число 10 преобразуется в строку
"7" * "4"        // => 28
                 // каждая строка преобразуется в число
var n = 1 - "x"; // => NaN
                 // строку "x" нельзя преобразовать
                 // в число
n +   объектов"  // => "NaN объектов"
                 // n равно NaN; значение NaN
                 // преобразуется в строку "NaN"
```

В табл. 2.2 приведены правила преобразования типов интерпретатором JavaScript. Полужирным шрифтом выделены неочевидные преобразования, которые многим покажутся удивительными. Пустые ячейки означают, что в преобразовании нет необходимости и оно не выполняется.

Таблица 2.2. Преобразования типов

Исходное значение	Преобразуется в...			
	строку	число	булево значение	объект
undefined	"undefined"	NaN	false	TypeError
null	"null"	**0**	false	TypeError
true	"true"	**1**		Boolean(true)
false	"false"	**0**		Boolean(false)
		0	**false**	String("")
"1.2"		1.2	true	String("1.2")

Окончание табл. 2.2

Исходное значение	Преобразуется в...			
	строку	число	булево значение	объект
"one"		NaN	true	String("one")
0	"0"		**false**	Number(0)
-0	"0"		**false**	Number(-1)
NaN	"NaN"		**false**	Number(NaN)
Infinity	"Infinity"		true	Number(Infinity)
-Infinity	"-Infinity"		true	Number(-Infinity)
1 (конечное ненулевое число)	"1"		true	Number(1)
{} (любой объект)	toString()	toString() или valueOf()	true	
[] (пустой массив)		0	true	
[9] (один числовой элемент)	"9"	9	true	
['a'] (любой нечисловой массив)	Используется функция join()	NaN	true	
function() {} (любая функция)	Исходный код функции	NaN	true	

Интерпретатор JavaScript гибко преобразует типы значений, поэтому поведение оператора проверки на равенство (==) основано на таких же гибких представлениях о равенстве. Например, все приведенные ниже выражения возвращают `true`.

```
null == undefined
"0" == 0      // Перед сравнением строка
              // преобразуется в число
0 == false    // Перед сравнением булево значение
              // преобразуется в число
"0" == false  // Перед сравнением оба операнда
              // преобразуются в числа
```

В большинстве случаев преобразование типов JavaScript выполняется автоматически, но иногда полезно задать явное преобразование (например, для того чтобы сделать код понятнее).

Выполнить явное преобразование типов проще всего с помощью функций `Boolean()`, `Number()`, `String()` и `Object()`.

```
Number("3")   // => 3
String(false) // => "false"  Можно также
              // написать false.toString()
Boolean([])   // => true
Object(3)     // => new Number(3)
```

Любое значение, отличное от `null` или `undefined`, поддерживает метод `toString()`, возвращающий строковое представление значения. Обычно результат тот же, что и возвращаемый функцией `String()`.

Некоторые операторы JavaScript, выполняющие неявное преобразование типов, иногда используются специально для преобразования. Если в операторе + один из операндов является строкой, то другой операнд всегда преобразуется в строку. Унарный оператор + преобразу-

ет свой операнд в число. Унарный операнд ! преобразует свой операнд в булев тип и возвращает инвертированное значение. Ниже приведены типичные выражения, которые часто можно встретить в реальных программах.

```
x +    // Эквивалент String(x)
+x     // Эквивалент Number(x); можно также x-0
!!x    // Эквивалент Boolean(x)
```

Форматирование и синтаксический анализ чисел — типичные задачи в JavaScript-программах, поэтому для их решения предоставлены специальные функции, позволяющие жестко контролировать процесс преобразования чисел в строки и наоборот.

Метод `toString()`, определенный в классе `Number`, принимает необязательный аргумент, задающий основание системы счисления. Если не задать этот аргумент, преобразование будет выполнено в десятичном счислении. Можно задавать основания от 2 до 36.

```
var n = 17;
binary_string = n.toString(2);       // => "10001"
octal_string = "0" + n.toString(8); // => "021"
hex_string = "0x" + n.toString(16); // => "0x11"
```

При работе с научными и финансовыми данными часто желательно задавать количество десятичных позиций после разделителя или точность (количество значащих цифр) результата преобразования. Иногда желательно также задать вывод результата в экспоненциальном формате. Для этого в классе `Number` есть три метода.

```
var n = 123456.789;
n.toFixed(2);        // "123456.79"
n.toExponential(3); // "1.235e+5"
n.toPrecision(7);    // "123456.8"
```

При получении строки функция `Number()` пытается методами синтаксического анализа извлечь из нее целочисленный или действительный числовой литерал. Эта функция работает только с основанием счисления 10 и не принимает замыкающие символы, не являющиеся частью литерала. Глобальные функции `parseInt()` и `parseFloat()` более гибкие. Функция `parseInt()` синтаксически анализирует только целые числа, а функция `parseFloat()` — как целые, так и действительные. Если строка начинается с `"0x"` или `"0X"`, функция `parseInt()` интерпретирует ее как шестнадцатеричное число. Обе эти функции пропускают ведущие (т.е. расположенные в начале) пробельные символы, считывают все имеющиеся цифровые символы и пропускают все, что стоит после первого встретившегося нецифрового символа. Если первый непробельный символ не является частью числового литерала, эти функции возвращают `NaN`.

```
parseInt("3 мышки")         // => 3
parseFloat(" 3.14 метра")   // => 3.14
parseInt("-12.34")          // => -12
parseInt("0xFF")            // => 255
parseFloat("$72.47");       // => NaN
```

Функция `parseInt()` принимает необязательный второй аргумент, задающий основание системы счисления. Допустимы значения от 2 до 36.

```
parseInt("11", 2);   // => 3 (1*2 + 1)
parseInt("077", 8);  // => 63 (7*8 + 7)
parseInt("ff", 16);  // => 255 (15*16 + 15)
```

Объявление переменных

Перед использованием в программе переменную нужно *объявить*. Объявление переменной выполняется с помощью ключевого слова `var`.

```
var i;
var sum;
```

С помощью одного слова var можно объявить несколько переменных.

```
var i, sum;
```

При объявлении можно инициализировать переменные.

```
var message = "hello";
var i = 0, j = 0, k = 0;
```

Если при объявлении переменной с помощью ключевого слова var не задать начальное значение, переменная будет объявлена, но будет содержать значение undefined, пока программа явно не присвоит ей другое значение.

Слово var можно использовать в инструкциях for и for/in (см. главу 4), что позволяет кратко объявить переменную непосредственно в цикле.

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--) console.log(i*j);
for(var p in o) console.log(p);
```

В статически типизированных языках, таких как C или Java, ключевое слово var не имеет аналога, потому что в этих языках каждой переменной жестко присваивается определенный тип. В JavaScript переменная может содержать значение любого типа. Например, в JavaScript можно сначала присвоить переменной число, а затем присвоить этой же переменной строку.

```
var i = 10;
i = "десять";
```

Переменную можно объявить с помощью ключевого слова var несколько раз. Если при повторном объявлении

выполняется инициализация, то оно эквивалентно оператору присваивания.

При попытке прочитать значение необъявленной переменной интерпретатор JavaScript генерирует ошибку. Согласно ECMAScript 5 в строгом режиме ошибкой также считается попытка присвоить значение необъявленной переменной. Однако в нестрогом режиме и в старых интерпретаторах при этом создается переменная, которая считается свойством глобального объекта и работает почти так же, как и свойство объявленного глобального объекта. Это означает, что глобальные переменные можно не объявлять. Тем не менее это плохая привычка, служащая источником многих ошибок, поэтому рекомендуется всегда объявлять переменные с помощью ключевого слова `var`.

Область видимости переменной — это фрагмент исходного кода, в котором ее можно использовать, т.е. в котором она считается определенной. Глобальная переменная имеет глобальную область видимости; это означает, что она определена в любой точке программы. В то же время переменная, объявленная внутри функции, определена только в теле данной функции; это *локальная* переменная, имеющая ограниченную область видимости. Параметры функции также считаются локальными переменными, определенными только в теле функции.

Локальная переменная в теле функции обладает приоритетом над глобальной переменной с тем же именем. Как уже упоминалось, глобальную переменную можно не объявлять, но локальную переменную всегда нужно объявлять с помощью ключевого слова `var`. Определения функций могут быть вложенными, причем каждая функция имеет собственную локальную область видимости, поэтому допускается несколько вложенных слоев локальных областей видимости.

В языках семейства C каждый блок кода в фигурных скобках имеет собственную область видимости, и переменная не видна за пределами блока, в котором она объявлена. Такое поведение называется *блочной видимостью*. Однако в JavaScript используется не блочная, а *функциональная видимость*. Это означает, что переменная видима внутри функции, в которой она объявлена, а также внутри каждой вложенной функции любого уровня вложенности.

Кроме того, функциональная видимость означает, что переменная видима во всем теле функции, в которой она объявлена, в том числе в коде, расположенном до объявления. Фактически код ведет себя так, как будто все инструкции объявления расположены в начале функции.

Выражения и операторы

Выражение – это фрагмент или фраза программы, которую интерпретатор может вычислить, вернув ее значение. Наиболее простое выражение – числовой или строковый литерал. Имя переменной – это тоже выражение, возвращающее значение переменной. Сложные выражения состоят из простых. Например, выражение доступа к элементу массива состоит из его имени, которое возвращает адрес массива, и квадратных скобок с индексом, которые возвращают целое число – номер элемента массива. После всех этих манипуляций возвращается значение, хранящееся в ячейке массива с данным номером. Еще пример: выражение вызова функции состоит из двух выражений – имени, возвращающего объект функции, и нескольких параметров, передаваемых функции в качестве аргументов (их количество может быть нулевым).

Чаще всего составное выражение получается из двух простых с помощью *оператора*. Оператор принимает несколько *операндов* (обычно два) и вычисляет результирующее значение. Например, в составном выражении `x*y` используется оператор умножения. Он вычисляет произведение выражений `x` и `y`. Вместо "вычисляет значение" часто пишут "оператор возвращает значение".

Выражения

Простейшее выражение (иногда его называют *первичным*) не содержит других, более простых выражений. В JavaScript первичным выражением может быть константа (литерал), ссылка на переменную или ключевое слово.

Литералы — это неизменные значения, внедренные непосредственно в программу. Ниже приведены примеры литералов.

```
1.23          // Числовой литерал
"Привет!"     // Строковый литерал
/pattern/     // Литерал регулярного выражения
```

Такие ключевые слова, как `true`, `false`, `null` и `this`, являются первичными выражениями.

И наконец, рассмотрим несколько примеров первичных выражений третьего типа: имен переменных.

```
i   // Возвращает значение переменной i
sum // Возвращает значение переменной sum
```

Встретив в коде идентификатор, интерпретатор JavaScript предполагает, что это имя переменной, и возвращает ее значение. Если переменной с таким именем не существует, выражение возвращает значение `undefined`. Однако в строгом режиме, определенном в ECMAScript 5, попытка извлечь значение несуществующей переменной приводит к генерации ошибки `ReferenceError`.

Инициализаторы

Инициализатор объекта или массива — это выражение, возвращаемым значением которого является создаваемый объект или массив. Выражения инициализаторов иногда называют *объектными литералами* или *литералами массивов*. Однако в отличие от истинных литералов они не являются первичными выражениями, потому что содержат вложенные выражения, определяющие значения свойств и элементов.

Инициализатор массива представляет собой заключенный в квадратные скобки список выражений, разделенных запятыми. Значением инициализатора массива

служит создаваемый массив. Элементы нового массива получают значения, указанные в списке.

```
[]        // Пустой массив
[1+2,3+4] // Массив из двух элементов
```

В инициализаторе массива выражения элементов сами могут быть инициализаторами массивов. Следовательно, в коде можно создавать вложенные массивы.

```
var matrix = [[1,2,3], [4,5,6], [7,8,9]];
```

После выражения, расположенного последним, можно поместить еще одну запятую.

Выражения объектных инициализаторов напоминают выражения инициализаторов массивов, но вместо квадратных скобок в них используются фигурные. Кроме того, каждое вложенное выражение содержит префикс, состоящий из имени свойства и двоеточия.

```
var p = { x:2, y:1 }; // Объект с двумя свойствами
var q = {};           // Пустой объект без свойств
q.x = 2; q.y = 1;     // Теперь q имеет те же
                      // свойства, что и p
```

Объектные литералы могут быть вложенными.

```
var rectangle = { upperLeft: { x: 2, y: 2 },
                  lowerRight: { x: 4, y: 5 } };
```

Выражения в инициализаторе объекта или массива вычисляются при каждом обращении к инициализатору. Кроме того, эти выражения не обязательно должны быть константами; они могут быть любыми выражениями языка. В качестве имен свойств в объектных литералах можно использовать не идентификаторы, а строки, заключенные в двойные кавычки. Это полезно при задании имен свойств, которые являются зарезервированными словами и не могут быть идентификаторами.

```
var side = 1;
var square = { "ul": { x: p.x, y: p.y },
               'lr': { x: p.x + side,
                       y: p.y + side}};
```

Обращение к свойствам

Выражение обращения к свойству возвращает значение свойства объекта или элемента массива. В JavaScript определены две формы обращения к свойству.

выражение.*идентификатор*
выражение [*выражение*]

В первой форме выражение задает объект, а идентификатор — имя нужного свойства. Во второй форме первое выражение задает объект или массив, а второе выражение, приведенное в квадратных скобках, задает имя нужного свойства или индекс элемента массива. Рассмотрим ряд конкретных примеров.

```
var o = {x:1,y:{z:3}}; // Создание объекта
var a = [o,4,[5,6]];   // Создание массива
o.x          // => 1: свойство x объекта o
o.y.z        // => 3: свойство z выражения o.y
o["x"]       // => 1: свойство x объекта o
a[1]         // => 4: элемент 1 выражения a
a[2]["1"]    // => 6: элемент 1 выражения a[2]
a[0].x       // => 1: свойство x выражения a[0]
```

Форма *.идентификатор* — более простая из двух форм доступа к свойствам, но ее можно использовать, только когда у свойства есть имя, являющееся допустимым идентификатором, причем это имя известно при написании кода. Если имя свойства является ключевым словом или содержит пробелы либо знаки пунктуации, необходимо использовать форму с квадратными скобками. Кроме того, форму с квадратными скобками необходимо использовать при обращении к элементам массивов в случае,

когда имя свойства заранее неизвестно и вычисляется на этапе выполнения кода.

Определение функции

Значением выражения, определяющего функцию, служит сама определяемая функция. Фактически выражение определения функции является "функциональным литералом" в том же смысле, что и инициализатор объекта является "объектным литералом". Обычно выражение определения функции состоит из ключевого слова `function`, после которого расположен список идентификаторов (имен параметров), разделенных запятыми, и блока кода JavaScript (тела функции) в фигурных скобках. Ниже приведен пример определения функции.

```
// Эта функция возвращает квадрат аргумента
var square = function(x) { return x * x; }
```

Функцию можно также определить с помощью инструкции, а не выражения, как будет показано в главе 7.

Вызов функции

Выражение вызова определяется синтаксисом JavaScript и начинается с выражения функции (обычно это ее имя), которое идентифицирует вызываемую функцию и после которого находится список аргументов в скобках. Аргументы являются выражениями и отделены друг от друга запятыми. Ниже приведен ряд примеров вызова функций.

```
f(0)           // Функция f с аргументом 0
Math.max(x,y,z) // Функция Math.max
               // с аргументами x, y и z
a.sort()       // Функция a.sort без аргументов
```

При вычислении выражения вызова сначала вычисляется выражение функции, а затем — выражения аргумен-

тов, в результате чего генерируется список значений аргументов. Если значение выражения функции не является функцией, генерируется ошибка `TypeError`. Значения аргументов присваиваются в заданной последовательности именам параметров, указанным в определении функции. После этого выполняется тело функции. Если в функции есть инструкция `return`, то вычисляемое ею значение становится значением выражения вызова. В противном случае выражение вызова получает значение `undefined`.

Каждое выражение вызова содержит некоторое выражение и пару скобок со списком (возможно, пустым). Если перед скобками находится выражение доступа к свойству, то выполняется *вызов метода*. В инструкции вызова метода объект или массив, содержащий вызываемое свойство, становится значением параметра `this`, а тело функции выполняется. Этим обеспечивается соответствие синтаксиса JavaScript парадигме объектно-ориентированного программирования, согласно которой функции (в объектно-ориентированном программировании они называются *методами*) принадлежат конкретным объектам и выполняются путем обращения к этим объектам.

Создание объекта

Выражение создания объекта вызывает функцию, называемую *конструктором*, для инициализации свойств объекта. Выражения создания объекта напоминают выражения вызова функций за исключением того, что перед ними находится ключевое слово `new`.

```
new Object()
new Point(2,3)
```

Если в выражении создания объекта конструктору не передается ни один аргумент, пустую пару скобок можно опустить.

```
new Object
new Date
```

Когда вычисляется выражение создания объекта, интерпретатор JavaScript сначала создает пустой объект, такой же, как и создаваемый инициализатором в блоке, ограниченном фигурными скобками {}. Затем интерпретатор вызывает определенную функцию с заданными аргументами и передает ей новый объект как значение ключевого слова this. В теле функции ключевое слово this можно использовать для инициализации свойств создаваемого объекта. Функции, предназначенные для использования в качестве конструкторов, не возвращают значения. Значением для выражения создания объекта является сам созданный и инициализированный объект. Если конструктор возвращает объектное значение, оно становится значением для выражения создания объекта, а созданный объект уничтожается.

Операторы

Операторы используются в арифметических и логических выражениях, а также в выражениях сравнения и присваивания. Полный список операторов JavaScript приведен в табл. 3.1.

Таблица 3.1. Операторы JavaScript

Оператор	Операция	Типы
++	Префиксный или постфиксный инкремент	lval⊛num
	Префиксный или постфиксный декремент	lval⊛num
	Отрицание числа	num⊛num

Продолжение табл. 3.1

Оператор	Операция	Типы
+	Преобразование в число	num@num
	Обращение битов	int@int
!	Обращение булева значения	bool@bool
delete	Удаление свойства	lval@bool
typeof	Вычисление типа операнда	Любой@str
void	Возврат неопределенного значения	Любой@undef
*, /, %	Умножение, деление, остаток деления	num, num@num
+, -	Сложение, вычитание	num, num@num
+	Конкатенация строк	str, str@str
<<	Сдвиг влево	int, int@int
>>	Сдвиг вправо со знаком	int, int@int
>>>	Сдвиг вправо с нулем	int, int@int
<, <=, >, >=	Сравнение по величине	num, num@bool
<, <=, >, >=	Сравнение по алфавиту	str, str@bool
instanceof	Проверка класса объекта	obj, func@bool
in	Проверка существования свойства	str, obj@bool
==	Проверка на равенство	Любой, любой@bool
!=	Проверка на неравенство	Любой, любой@bool
===	Проверка на строгое равенство	Любой, любой@bool
!==	Проверка на строгое неравенство	Любой, любой@bool
&	Побитовая конъюнкция AND	int, int@int
	Побитовая исключающая дизъюнкция XOR	int, int@int

Окончание табл. 3.1

Оператор	Операция	Типы
|	Побитовая дизъюнкция OR	int, int@int
&&	Конъюнкция AND	Любой, любой ® любой
||	Дизъюнкция OR	Любой, любой ® любой
?:	Выбор из двух операндов	bool, любой, любой ® любой
=	Присваивание	lval, любой ® любой
*=, /=, %=, +=, -=, &=, ^-, |=, <<=, >>=, >>>=	Операция и присваивание	lval, любой ® любой
	Отбрасывание первого и возвращение второго операнда	Любой, любой ® любой

Операторы, перечисленные в табл. 3.1, отсортированы в порядке убывания приоритета. Некоторые однотипные операторы, расположенные в смежных ячейках таблицы (например, операторы сравнения), имеют одинаковые приоритеты. От приоритетов зависит последовательность выполнения операций. Если два оператора находятся в коде рядом, то сначала выполняется операция с более высоким приоритетом (в верхней части табл. 3.1), а затем — с более низким.

Рассмотрим следующее выражение.

```
w = x + y*z;
```

Оператор умножения * имеет более высокий приоритет, чем оператор сложения +, поэтому сначала выполняется умножение, а затем — сложение. Кроме того, оператор = имеет самый низкий приоритет, поэтому присваивание выполняется после выполнения всех операций, указанных справа от знака равенства.

Последовательность выполнения операторов можно явно переопределить с помощью скобок. Чтобы в предыдущем примере сначала выполнялся оператор сложения, нужно записать инструкцию следующим образом.

```
w = (x+y) * z;
```

Важно помнить, что выражения доступа к свойству и вызова функции обладают более высоким приоритетом, чем любой из операторов, приведенных в табл. 3.1. Рассмотрим следующее выражение.

```
typeof my.functions[x](y)
```

Оператор `typeof` имеет один из самых высоких приоритетов, тем не менее в данном выражении операция `typof` выполняется после обоих обращений к свойствам и вызова функции.

На практике, если вы не уверены, какие операции будут выполняться в первую очередь, проще всего явно задать последовательность их выполнения с помощью скобок. Помнить все приоритеты необязательно, потому что порядок выполнения легко задать явно; к тому же код тогда будет намного понятнее. Полезно помнить только, что умножение и деление имеют более высокий приоритет, чем сложение и вычитание, и, кроме того, что оператор присваивания имеет один из самых низких приоритетов, поэтому почти всегда выполняется в последнюю очередь.

Некоторые операторы принимают значения любого типа, но для большинства операторов нужно, чтобы операнды имели определенный тип. Кроме того, почти все операторы возвращают значение определенного типа. В последнем столбце табл. 3.1 перед стрелкой указаны типы операндов, а после стрелки — тип результирующего (возвращаемого) значения. Количество типов перед стрелкой указывает на *арность* (количество аргументов,

или, что то же самое, операндов) оператора. *Унарные* операторы имеют один операнд, *бинарные* – два и *тернарные* – три. В JavaScript есть только один тернарный оператор – `?:`.

Некоторые операторы выполняют разные операции в зависимости от типов операндов. Фактически это разные операторы, обозначаемые одинаковыми символами. Наиболее характерный пример – оператор +, который выполняет суммирование после получения числовых операндов или конкатенацию после получения строк. Аналогично операторы сравнения, такие как <, получив числа, сравнивают их по величине, а получив строки, сравнивают их по алфавиту.

Ряд операторов ожидают операнды `lval` (left value – значение слева). Фактически `lval` – это не тип, а традиционное обозначение любого выражения, которое может находиться с левой стороны оператора присваивания. Значениями `lval` в JavaScript являются переменные, свойства объектов и элементы массивов.

Вычисление простого изолированного выражения (например, `2*3`) не влияет ни на состояние программы, ни на любые последующие операции. Однако вычисление некоторых выражений приводит к *побочным эффектам* и может затронуть результаты будущих операций. Наиболее очевидный пример – операторы присваивания, которые изменяют значения переменных и свойств. Это же справедливо для операторов инкремента `--` и декремента `++`, поскольку они неявно изменяют значения переменных и свойств. Побочными эффектами обладает также оператор `delete`, потому что он делает значение свойства неопределенным (но это не то же самое, что присвоение значения `undefined`).

Арифметические операторы

В этом разделе рассматриваются операторы, выполняющие арифметические и другие числовые операции над своими операндами.

- **Умножение (*).** Вычисление произведения двух операндов.
- **Деление (/).** Деление первого операнда на второй. В языках программирования, различающих целые и действительные числа, результат деления двух целочисленных операндов также целочисленный. Однако в JavaScript все числа действительные, поэтому результат деления всегда действительный. Например, операция `5/2` возвращает `2.5`, а не `2`. Деление на нуль возвращает положительную или отрицательную бесконечность, а деление нуля на нуль — значение `NaN`. Ни в одном из этих случаев ошибка не генерируется.
- **Деление по модулю (%).** Вычисление остатка целочисленного деления первого операнда на второй. Знак результата совпадает со знаком первого операнда. Например, операция `5%2` возвращает `1`, а `-5%2` возвращает `-1`. Обычно оператор деления по модулю используется с целочисленными операндами, но в JavaScript его можно применять с действительными значениями. Например, оператор `6.5%2.1` возвращает `0.2`.
- **Сложение (+).** Бинарный оператор + возвращает сумму двух операндов, если они числовые. Если же операнды являются строками, он выполняет их конкатенацию.

```
1 + 2                        // => 3
"Привет" +     + "всем!"     // => "Привет всем!"
"1" + "2"                    // => "12"
1 + 2 + " белых мышей";      // => "3 белых мышей"
1 + (2 + " белых мышей");    // => "12 белых мышей"
```

Когда значения обоих операндов — числа или строки, результат очевиден. В противном случае (когда один операнд — число, а другой — строка) выполняется преобразование типов, и результат операции зависит от того, в какой тип преобразуется один из операндов. Правила преобразования для оператора + отдают приоритет конкатенации строк: если один из операндов является строкой или объектом, преобразуемым в строку, то другой операнд преобразуется в строку и выполняется конкатенация. Сложение выполняется, только если ни один из операндов не является строкой и не преобразуется в нее.

- **Вычитание (–).** Вычитание значения правого операнда из значения левого операнда.

Кроме перечисленных выше бинарных операторов, язык JavaScript определяет ряд унарных арифметических операторов. Унарный оператор изменяет значение одного операнда.

- **Унарный плюс (+).** Преобразует свой операнд в число (или значение `NaN`) и возвращает преобразованное значение. Если операндом является число, то этот оператор ничего не делает.
- **Унарный минус (–).** Преобразует операнд в число и при необходимости изменяет знак результата.
- **Инкремент (++).** Увеличивает операнд на единицу. Операнд должен быть значением `lval` (т.е. пере-

менной, элементом массива или свойством объекта). Если операнд не является числом, то этот оператор преобразует его в число, добавляет единицу и присваивает увеличенное значение обратно переменной, элементу или свойству.

Возвращаемое значение оператора ++ зависит от его позиции относительно операнда. Если он находится перед операндом (*префиксный инкремент*), то увеличивает операнд на единицу и возвращает увеличенное значение. Если же он находится после операнда (*постфиксный инкремент*), то увеличивает операнд на единицу, но возвращает не увеличенное, а исходное значение. Рассмотрим различия между следующими двумя инструкциями.

```
var i = 1, j = ++i; // i и j равны 2
var i = 1, j = i++; // i равно 2, j равно 1
```

Этот оператор, как в префиксной, так и в постфиксной формах, часто используется для увеличения переменной счетчика в цикле `for`.

- **Декремент (–).** Принимает значение `lval`, преобразует его в число, вычитает из него единицу и присваивает результат обратно операнду. Как и оператор ++, возвращаемое значение зависит от позиции оператора относительно операнда. Находясь перед операндом (префиксная форма), он возвращает уменьшенное значение. Если же он расположен после операнда (постфиксная форма), то возвращает исходное значение.

Побитовые операторы манипулируют отдельными битами двоичного представления числа. В программах на JavaScript они используются редко. Если вы не знакомы

с двоичными представлениями десятичных целых чисел, можете пропустить этот раздел. Побитовые операторы принимают целочисленные операнды и обрабатывают их так, как будто это значения, представленные 32-разрядными целыми числами, а не действительные 64-разрядные значения, каковыми они на самом деле являются. При необходимости побитовые операторы сначала преобразуют операнд в число, а затем приводят полученное число к 32-разрядному целому путем отбрасывания дробной части и всех битов после 32-го. Для операторов сдвига необходим операнд с заполненными ячейками от 0 до 31.

- **Побитовая конъюнкция (&).** Другие названия — AND и И. Оператор `&` выполняет булеву операцию конъюнкции над каждым битом целочисленных аргументов. Бит результата равен 1, только если соответствующие биты в обоих операндах также равны 1. Например, выражение `0x1234 & 0X00FF` возвращает значение `0x0034`.

- **Побитовая дизъюнкция (|).** Другие названия — OR и ИЛИ. Оператор `|` выполняет булеву операцию дизъюнкции над каждым битом целочисленных аргументов. Бит результата равен единице, если единице равен хотя бы один из соответствующих битов операндов. Например, выражение `0x1234 | 0x00FF` возвращает `0x12FF`.

- **Побитовая исключающая дизъюнкция (^).** Другие названия — XOR и "Исключающее ИЛИ". Этот оператор выполняет исключающую дизъюнкцию над каждым битом целочисленных аргументов. Бит результата равен 1, если один из битов операндов (но не оба) равен 1. Например, выражение `0xFF0 ^ 0xF0F0` возвращает `0x0FF0`.

- **Побитовое отрицание (~).** Этот унарный оператор, помещаемый перед целочисленным операндом, инвертирует все биты операнда. Целочисленные значения со знаком представляются в JavaScript таким образом, что применение оператора ~ эквивалентно изменению знака числа и вычитанию из него единицы. Например, выражение `~0x0F` возвращает `0xFFFFFFF0` или `-16`.

- **Сдвиг влево (<<).** Этот оператор перемещает влево все биты первого операнда на расстояние, заданное вторым операндом. Например, после выполнения операции `a << 1` первый бит становится вторым, второй — третьим и т.д. В позицию первого, "ушедшего" бита записывается нуль, а значение последнего, 32-го, т.е. самого старшего бита, теряется. Важно отметить, что побитовый сдвиг влево на одну позицию эквивалентен умножению на 2, сдвиг на две позиции — умножению на 4 и т.д. Например, выражение `7 << 2` возвращает `28`.

- **Сдвиг вправо со знаком (>>).** Этот оператор перемещает все биты первого операнда вправо на расстояние, заданное вторым операндом, который должен быть целым числом от 0 до 31. Биты, сдвинутые дальше правой границы, теряются. Биты слева заполняются в зависимости от знака исходного первого операнда таким образом, чтобы сохранить знак числа. Если исходный операнд положительный, биты слева заполняются нулями, а если отрицательный — единицами. Сдвиг вправо на один бит эквивалентен делению на 2 с отбрасыванием остатка деления, сдвиг на два бита — делению на 4 и т.д. Например, выражение `7 >> 1` возвращает `3`, а `-7 >> 1` возвращает `-4`.

- **Сдвиг вправо с заполнением нулями (>>>).** Работает так же, как и оператор >>, за исключением того, что биты слева при сдвиге всегда заполняются нулями, независимо от знака первого операнда. Например, `-1 >> 4` возвращает -1, но `-1 >>> 4` возвращает `0x0FFFFFFF`.

Операторы сравнения

Эти операторы проверяют отношения между двумя значениями (такие, как "равно", "больше", "является свойством" и др.) и возвращают значение `true` или `false` в зависимости от результатов проверки. Выражения сравнения всегда возвращают булево значение. Это часто применяется в инструкциях `if`, `while` и `for` для управление ветвлением программы.

В JavaScript поддерживаются операторы `=`, `==` и `===`. Важно понимать различия между ними, потому что это три совершенно разные операции: присваивание, приверка на равенство и приверка на строгое равенство. В каждом конкретном случае необходимо выбирать правильный оператор. Иногда их называют одним и тем же словом "равно", однако рекомендуется даже мысленно называть их разными словами: "присвоить", "равно" и "строго равно".

- **Строгое равенство (===).** Другое название – оператор *сравнения на идентичность*. При проверке на равенство используется строгое определение идентичности, не включающее преобразования типов операндов. Если операнды имеют разные типы, оператор считает, что они не равны. Если оба операнда имеют примитивные типы и содержат одинаковые значения, они считаются равными. Кроме того, два операнда считаются равными, если оба

ссылаются на один и тот же объект, массив или функцию. Если они ссылаются на разные объекты, они не равны, даже если эти объекты имеют идентичные свойства. Аналогично два массива, состоящие из одних и тех же элементов, расположенных в одной и той же последовательности, не считаются равными. Важно учитывать также, что оператор проверки на строгое равенство считает значение `NaN` не равным никакому другому значению, включая само себя.

- **Строгое неравенство (!==).** Этот оператор — прямая противоположность оператору проверки на строгое равенство: он возвращает `false`, если два операнда строго равны, и `true` в противном случае.
- **Равенство (==).** Напоминает оператор проверки на строгое равенство, но менее "жесткий". Получив операнды разных типов, этот оператор пытается привести их к одному типу и сравнить преобразованные значения. Данный оператор считает значения `null` и `undefined` равными. Во многих случаях такое поведение полезно, но иногда преобразование может приводить к неожиданным сюрпризам. Все приведенные ниже выражения возвращают `true`.

```
1 == "1"
true == 1
"1" == true
false == 0
[] == 0
```

- **Неравенство (!=).** Этот оператор — полная противоположность оператору проверки на равенство: он возвращает `false`, если два значения нестрого равны, и `true` в противном случае.

Операторы проверки на неравенство вычисляют взаимное расположение двух операндов один относительно другого в некоторой "канонической" последовательности. Операнды, не являющиеся числами или строками, преобразуются в числа или строки. Строки интерпретируются как наборы 16-битовых целых чисел, и при сравнении строк фактически сравниваются их числовые значения. Важно отметить, что процедура сравнения строк чувствительна к регистру, причем все буквы ASCII в верхнем регистре считаются "меньшими", чем все буквы в нижнем регистре. Если не учитывать этого обстоятельства, можно получить неожиданные результаты. Например, выражение `"Paris" < "arc"` возвращает `true`, хотя буква `a` в алфавите первая.

- **Меньше (<).** Этот оператор возвращает `true`, если первый операнд меньше второго. В противном случае возвращается `false`.
- **Больше (>).** Возвращает `true`, если первый операнд больше второго. В противном случае возвращается `false`.
- **Меньше или равно (<=).** Возвращает `true`, если первый операнд меньше второго или равен ему. В противном случае возвращается `false`.
- **Больше или равно (>=).** Возвращает `true`, если первый операнд больше второго или равен ему. В противном случае возвращается `false`.

Два оператора сравнения представлены ключевыми словами `in` и `instanceof`.

- **Свойство существует (`in`).** Этот оператор принимает два операнда. Первый операнд должен быть строкой или иметь тип, который можно преобразо-

вать в строку. Второй операнд должен быть объектом. Оператор `in` возвращает `true`, если значение первого операнда является именем свойства правого операнда. Ниже приведен ряд примеров.

```
var p = { x:1, y:1 };
"x" in p          // => true
"z" in p          // => false
"toString" in p   // => true,
                  // так как p наследует toString
var a = [7,8,9];
"0" in a          // => true, так как a имеет элемент "0"
1 in a            // => true,
                  // так как число преобразуется в строку
```

- **Тип объекта (`instanceof`).** Оператор `instanceof` ожидает два операнда: слева — объект, а справа — класс объекта. Данный оператор возвращает `true`, если левый операнд является экземпляром класса, указанного в правом операнде. В противном случае возвращается `false`. В главе 8 будет показано, что в JavaScript класс объекта определяется функцией конструктора, которая инициализирует его. Следовательно, правый операнд оператора `instanceof` должен быть функцией. Ниже приведен ряд примеров.

```
var d = new Date();
d instanceof Date;   // => true
d instanceof Object; // => true
d instanceof Number; // => false
var a = [1, 2, 3];
a instanceof Array;  // => true
a instanceof Object; // => true
```

Логические выражения

Логические операторы `&&`, `||` и `!` реализуют правила булевой алгебры и часто используются совместно с опе-

раторами сравнения, позволяя объединять два выражения сравнения в одно, более сложное выражение. Важно помнить, что в JavaScript значения `null`, `undefined`, `0`, `""` и `NaN` считаются такими, которые имеют булево значение `false`. Все другие значения, включая объекты, строки и массивы, имеют булево значение `true`.

Оператор `&&` можно интерпретировать по-разному. На простейшем уровне при использовании с булевыми операндами этот оператор выполняет логическую операцию конъюнкции двух булевых значений (иногда эту операцию называют логическим AND или И). Оператор `&&` принимает два операнда и возвращает значение `true`, если оба операнда равны `true`. Если хотя бы один операнд равен `false`, возвращается значение `false`.

Часто оператор `&&` используется совместно с двумя выражениями сравнения.

```
// Возвращает true, если x=0 и y=0
x == 0 && y == 0
```

Любое выражение сравнения всегда равно либо `true`, либо `false`, поэтому оператор `&&` работает с ними так же, как и с булевыми значениями, возвращая `true` или `false`. Операторы сравнения имеют более высокий приоритет, чем операторы `&&` и `||`, поэтому приведенное выше выражение можно писать без скобок.

Однако для оператора `&&` не обязательно, чтобы его операнды были булевыми значениями. Он принимает любые значения, поскольку они могут быть преобразованы в булевы. Если оба операнда преобразуются в значение `true`, оператор `&&` возвращает непреобразованное значение (не обязательно `true`). Если же хотя бы один из операндов или оба преобразуются в значение `false`, оператор возвращает `false` или непреобразованное значение, дающее `false`. В JavaScript любое выражение (или инструкция), ожидаю-

щее булева значения, без проблем принимает любое значение, потому что оно автоматически преобразуется в булево. Важно также учитывать, что оператор && не всегда возвращает значение true или false (см. далее).

Таким образом, при получении значения, отличного от булева, оператор && сначала преобразует его в булево значение. Прежде всего преобразованию подвергается первый операнд, расположенный слева. Если результат преобразования равен false, значит, результат выражения равен false или непреобразованному значению независимо от значения второго операнда. Поэтому вычисление и преобразование второго операнда не выполняется. Во многих случаях важно учитывать это правило, потому что при вычислении второго операнда могут выполняться операции, влияющие на состояние программы (например, могут изменяться значения переменных).

С другой стороны, если результат преобразования левого операнда равен true, значит, результат выражения зависит от правого операнда. Если его значение равно true, то оператор && возвращает true или исходное значение, в противном случае он возвращает false. Рассмотрим ряд примеров.

```
var o = { x   1 };
var p = null;
o && o.x // => 1: o возвращает true, поэтому
         // выражение возвращает o.x
p && p.x // => null: p равно false, поэтому p.x не
         // вычисляется и ошибка не генерируется
```

Важно помнить, что оператор && может не вычислять правый операнд. В приведенном выше примере переменная p имеет значение null. Следовательно, при вычислении выражения p.x должна быть сгенерирована ошибка TypeError. Однако правый операнд не вычисляется, поэтому ошибка не возникает.

Оператор || вычисляет дизъюнкцию двух операндов (другие названия этой операции — OR и ИЛИ). Если хотя бы один или оба операнда преобразуются в true, он возвращает true или непреобразованное значение. Если же оба операнда преобразуются в false, он возвращает false или непреобразованное значение.

Чаще всего оператор || применяется с булевыми операндами, однако, как и оператор &&, он может использоваться с операндами других типов, и тогда его поведение может быть довольно сложным. Сначала он вычисляет левый операнд. Если его значение преобразуется в true, он возвращает true или непреобразованное значение (если операнд не булев). В противном случае вычисляется второй операнд и возвращается его значение.

Благодаря такому поведению оператор || иногда используется для выбора первого значения, преобразуемого в true. Рассмотрим приведенную ниже инструкцию.

```
var max = max_width||preferences.max_width||500;
```

Она работает следующим образом. Если определена переменная max_width, вычисление оператора прекращается и переменная max получает значение max_width. В противном случае анализируется свойство max_width объекта preferences. Если оно определено, то переменная max получает его значение. В противном случае переменной max присваивается значение 500.

Этот же прием часто используется в теле функции для установки значений параметров по умолчанию.

```
function copy(o, p) {
  // Если объект p не получен,
  // используется пустой объект
  p = p || {};
  // Тело функции
}
```

Унарный оператор ! инвертирует булево значение своего операнда. Например, если x равно или преобразуется в true, то !x возвращает false. Если же x равно или преобразуется в false, то выражение !x возвращает true. Оператор ! всегда возвращает true или false, поэтому его можно использовать для преобразования любого значения в булево, записав оператор дважды: !!x.

Поскольку ! является унарным оператором, он имеет очень высокий приоритет. Следовательно, чтобы инвертировать значение выражения p&&q, нужно применить скобки: !(p&&q).

Операторы присваивания

Оператор = присваивает заданное значение переменной, свойству объекта или элементу массива.

```
i = 0   // Переменной i присваивается значение 0
o.x = 1 // Свойству x объекта o
        // присваивается значение 1
```

Оператор = ожидает увидеть слева операнд lval, т.е. переменную, свойство объекта или элемент массива. Справа этот оператор ожидает произвольное значение любого типа. Значением выражения присваивания служит значение правого операнда. Оператор = присваивает его левому операнду.

Оператор присваивания обладает ассоциативностью справа налево. Это означает, что, если в выражении есть несколько операторов присваивания, они вычисляются начиная с правого. Следовательно, приведенный ниже код присваивает одно и то же значение нескольким переменным.

```
i = j = k = 0;
```

Кроме обычного оператора присваивания (=), в JavaScript определены несколько дополнительных операторов, являющихся сокращенными обозначениями двух операций: присваивания и одной из арифметических операций. Например, оператор += выполняет сложение и присваивание. Оператор

```
total += sales_tax
```

эквивалентен оператору

```
total = total + sales_tax
```

Как нетрудно догадаться, оператор += работает не только с числами, но и со строками. Получив числовые операнды, он выполняет сложение и присваивание, а получив строковые — конкатенацию и присваивание.

Аналогично работают операторы -=, *=, &= и др.

Интерпретация строк

Как и многие другие языки программирования, JavaScript может интерпретировать строки исходного кода, преобразуя их в значения. Для этого нужно вызвать в коде глобальную функцию `eval()`.

```
eval("3+2")  // => 5
```

Динамическое вычисление строк исходного кода — мощное средство языка, но необходимость в его практическом применении возникает очень редко. Каждый раз, используя функцию `eval()`, подумайте, действительно ли она необходима в данной ситуации (в приведенном выше примере она явно лишняя). Технически `eval()` является функцией, но она рассматривается в данном разделе, посвященном операторам, потому что во многом ведет себя, как оператор.

Функция `eval()` принимает один аргумент. Если передать ей любое значение, отличное от строки, она всего лишь вернет это же значение. Если же передать строку, она попытается синтаксически проанализировать ее и преобразовать в код JavaScript. Если ей не удастся сделать это, она сгенерирует ошибку `SyntaxError`. При успешном завершении синтаксического анализа функция вычисляет полученный код и возвращает значение последнего выражения или инструкции. Если последнее выражение или инструкция не имеет значения, функция `eval()` возвращает значение `undefined`.

Важная особенность функции `eval()` состоит в том, что она использует имена переменных и объектов текущей среды, определенные в коде. Функция находит значения переменных и определяет новые переменные и функции так же, как и в локальном коде. Если в коде определена локальная переменная `x` и вызвана функция `eval("x")`, она вернет текущее значение переменной `x`. Инструкция `eval("x=1")` изменяет значение локальной переменной `x`. Инструкция `eval("var y = 3;")` объявляет новую локальную переменную `y`. Аналогично этому в инструкции вызова можно даже определить новую локальную функцию.

```
eval("function f() { return x+1; }");
```

Чтобы интерпретатор JavaScript эффективно обрабатывал функцию `eval()`, на нее наложено существенное ограничение: она работает, как описано выше, только при вызове с исходным именем `eval`. Как и любой функции, ей можно присвоить другое имя, но при вызове под этим именем она будет работать иначе. В частности, она будет интерпретировать полученную строку как код глобального уровня. Таким способом можно объявлять новые глобальные переменные, но тогда невозможно бу-

дет использовать или изменять локальные переменные вызывающей функции.

В версиях браузеров IE8 и ниже при вызове с другим именем функция `eval()` не переходит на глобальный уровень. Эти браузеры определяют глобальную функцию `execScript()`, которая выполняет свой строковый аргумент как сценарий верхнего уровня. В отличие от `eval()`, функция `execScript()` всегда возвращает `null`.

Строгий режим, определенный в ECMAScript 5, накладывает на поведение функции `eval()` дополнительные ограничения. Когда функция `eval()` вызывается в строгом режиме или передаваемая строка начинается с директивы `use strict`, функция может запрашивать или устанавливать локальные переменные, но не может определять новые переменные и функции в локальной области видимости. Кроме того, строгий режим делает функцию еще более похожей на оператор тем, что в этом режиме `eval` является зарезервированным словом.

Дополнительные операторы

В JavaScript определен ряд других операторов, которые не относятся к описанным выше категориям.

Условный оператор ?:

В JavaScript это единственный тернарный оператор (т.е. оператор, имеющий три операнда). В книгах его часто обозначают как `?:`, хотя в коде он используется не в таком виде. Этот оператор имеет три операнда. Первый указывается перед символом `?`, второй — между символами `?` и `:`, а третий — после символа `:`.

```
x > 0 ? x   -x // Абсолютное значение x
```

Операнды условного оператора могут иметь любой тип. Первый операнд интерпретируется и вычисляется как булев. Если значение первого операнда может быть преобразовано в `true`, то вычисляется второй операнд, и оператор возвращает его значение. В противном случае, если первый операнд преобразуется в `false`, вычисляется и возвращается второй операнд. В любом случае вычисляется только один операнд: второй или третий, оба операнда никогда не вычисляются.

Того же результата несложно достичь с помощью инструкции `if`, однако оператор `?:` в простых ситуациях более удобен. Ниже приведен типичный пример его использования. Если свойство `user.name` существует и имеет значение, оно используется в результирующей строке. В противном случае в строку подставляется слово `всем`.

```
greeting = "Привет   +
    (user.name ? user.name   "всем");
```

Оператор `typeof`

Это унарный оператор, который записывается перед своим единственным операндом любого типа и возвращает строковое значение, содержащее тип операнда. Ниже приведены возвращаемые значения оператора `typeof` при получении значений разных типов.

`x`	`typeof x`
`undefined`	`"undefined"`
`null`	`"object"`
`true` **или** `false`	`"boolean"`
Любое число или `NaN`	`"number"`
Любая строка	`"string"`
Любая функция	`"function"`
Любой нефункциональный объект	`"object"`

Оператор typeof можно использовать, например, в следующем выражении.

```
(typeof value == "string") ?
        "'" + value + "'"    value
```

Оператор delete

Это унарный оператор, который пытается удалить указанное свойство объекта или элемент массива (в C++ есть ключевое слово delete, но оно имеет совершенно другой смысл, чем оператор с таким же именем в JavaScript). Как и многие другие операторы, оператор delete чаще всего используется не для получения возвращаемого значения, а ради побочных эффектов, в частности, для выполнения нужных операций. Ниже приведен ряд примеров его использования.

```
var o = {x:1, y:2}, a = [1,2,3];
delete o.x;    // Удаление свойства x
"x" in o       // => false: свойства не существует
delete a[2];   // Удаление последнего элемента
2 in a         // => false: этого элемента не существует
```

Оператор void

Это унарный оператор, записываемый перед единственным операндом любого типа. Данный оператор очень необычный и применяется редко. Он сначала вычисляет свой операнд, а затем отбрасывает его значение и возвращает значение undefined. Поскольку значение операнда отбрасывается, применять оператор void имеет смысл только ради его побочных эффектов.

Оператор "запятая"

Это бинарный оператор, принимающий операнды любого типа. Он вычисляет сначала левый операнд, а затем — правый, после чего возвращает значение правого операнда. Выражение слева всегда вычисляется, но его значение отбрасывается. Это означает, что применять данный оператор имеет смысл, только когда вычисление выражения слева приводит к полезным побочным эффектам. Чаще всего он используется в цикле `for` (рассматривается в главе 4), содержащем несколько переменных цикла. В приведенном ниже примере запятые служат для того, чтобы можно было манипулировать двумя переменными в одном выражении.

```
for(var i=0,j=10; i < j; i++,j--) console.log(i+j);
```

Инструкции

В главе 3 речь шла о выражениях JavaScript. В данной главе рассматриваются более крупные части кода — *инструкции*, которые могут содержать выражения. В коде JavaScript инструкции отделяются одна от другой точками с запятыми. Инструкция — это фрагмент программы, приказывающий компьютеру что-либо сделать.

Выражение, имеющее побочные эффекты, также приказывает компьютеру что-либо сделать, например присвоить переменной значение или вызвать функцию. Такое выражение может даже служить отдельной инструкцией. В этом случае она называется *инструкцией-выражением*. Но чаще инструкции состоят из многих выражений. Еще одна категория инструкций — *инструкции-объявления*, которые объявляют новые переменные или определяют новые функции.

Программа JavaScript представляет собой последовательность выполняемых инструкций. Интерпретатор JavaScript выполняет их одну за другой в последовательности их расположения в программе. Впрочем, последовательность может быть изменена с помощью *управляющих инструкций*, которые приводят к ветвлению программы. Управляющие инструкции делятся на три категории.

- **Условные инструкции.** Это такие инструкции, как `if` и `switch`. Они заставляют интерпретатор выполнять или пропускать определенные инструкции в зависимости от значения управляющего выражения.

- **Циклы.** Например, циклы `while` и `for`. Они предназначены для повторяющегося выполнения ряда инструкций.
- **Переходы.** Это такие инструкции, как `break`, `return` и `throw`. Они заставляют интерпретатор переходить из одной части программы в другую.

В табл. 4.1 приведен синтаксис инструкций JavaScript. Далее каждая инструкция рассматривается более подробно.

Таблица 4.1. Инструкции JavaScript

Инструкция	Синтаксис	Назначение
`break`	`break [`*имя_метки*`];`	Выход из внутреннего цикла, блока `switch` или именованного блока инструкций
`case`	`case` *выражение*`:`	Метка инструкции в блоке `switch`
`continue`	`continue [`*метка*`];`	Переход к началу следующей итерации внутреннего или именованного цикла
`debugger`	`debugger;`	Точка прерывания отладчика
`default`	`default:`	Метка инструкции, установленной по умолчанию в блоке `switch`
`do/while`	`do` *инструкция* `while(`*выражение*`);`	Альтернативный вариант цикла `while`
`empty`		Пустой оператор; никакая операция не выполняется
`for`	`for(`*инициализация*`;` *проверка*`;` *инкремент*`)` *выражение*	Цикл со счетчиком
`for/in`	`for(`*переменная* `in` *объект*`)` *инструкция*	Цикл с перечислением свойств объекта

Окончание табл. 4.1

Инструкция	Синтаксис	Назначение
function	function *имя_функции*([*параметр*[,...]]) {*тело_функции*}	Объявление и определение функции
if/else	if(*выражение*) *инструкция1* [else *инструкция2*]	Условный переход; ветвление программы
label	*имя_метки*: *инструкция*	Отметка места в коде
return	return [*выражение*];	Возврат значения из функции, завершение работы функции
switch	switch (*выражение*) {*инструкции*}	Выбор одного из вариантов ветвления в зависимости от значения выражения
throw	throw *выражение*;	Принудительная генерация исключения
try	try {*инструкции*} [catch {*инструкции*}] [finally {*инструкции*}]	Перехват и обработка исключения
use strict	"use strict"	Наложение ограничений строгого режима на сценарий или функцию
var	var *имя* [=*выражение*] [,...];	Объявление и инициализация одной или нескольких переменных
while	while(*выражение*) инструкция	Цикл с условием
with	with (*объект*) *инструкция*	Добавление объекта в начало цепочки областей видимости; в строгом режиме эта инструкция запрещена

Инструкция-выражение

Простейшая разновидность инструкции — выражение с побочными эффектами. Наиболее характерными при-

мерами инструкции-выражения служат выражения присваивания.

```
greeting = "Привет,  " + name;
i *= 3;
```

Выражения с операторами инкремента (++) и декремента (--) также могут быть инструкциями присваивания. Их побочный эффект состоит в изменении значения переменной.

```
counter++;
```

Оператор `delete` имеет важный побочный эффект — он удаляет свойство объекта. Поэтому он чаще используется в качестве инструкции, а не как часть большего выражения.

```
delete o.x;
```

Вызов функции — еще одна категория инструкций-выражений.

```
alert(greeting);
window.close();
```

Эти вызовы функций являются выражениями, но они приводят к побочным эффектам, влияющим на браузер, поэтому они могут использоваться в качестве инструкций.

Составные и пустые инструкции

Блок объединяет множество инструкций в одну *составную инструкцию*. Блок — это последовательность инструкций, заключенная в фигурные скобки. Следовательно, приведенный ниже код работает как одна инструкция, и его можно использовать в любом месте программы на JavaScript, кроме отдельной инструкции.

```
{
  x = Math.PI;
  cx = Math.cos(x);
  console.log("cos(PI) = " + cx);
}
```

Объединение инструкций в большие блоки используется в программировании на JavaScript довольно часто. Выражения могут содержать вложенные выражения, точно так же инструкции могут содержать вложенные инструкции. Формально синтаксис JavaScript обычно разрешает создать только одну вложенную инструкцию. Например, синтаксис цикла `while` определяет, что телом цикла может быть только одна инструкция или блок инструкций. Используя блоки инструкций, можно включать много инструкций в такое место кода, в котором разрешена только одна инструкция.

Составные инструкции позволяют применить много инструкций там, где синтаксис JavaScript допускает одну инструкцию. Противоположный случай — *пустая инструкция*. Она позволяет не включить ни одной инструкции в место, в котором ожидается одна инструкция.

Формально пустая инструкция представляет собой всего лишь точку с запятой. Встретив пустую инструкцию, интерпретатор JavaScript не выполняет никаких действий. Пустая инструкция часто полезна для создания цикла с пустым телом.

```
// Инициализация элементов массива
for(i = 0; i < a.length; a[i++] = 0);
```

Инструкция-объявление

Ключевые слова `var` и `function` используются в *инструкциях-объявлениях*, которые объявляют или определяют переменные и функции. Эти инструкции определяют идентификаторы (имена переменных и функций),

которые в результате можно использовать в других местах программы, в частности, можно присваивать им значения. Инструкции-объявления не выполняют никаких операций над данными, но, создавая переменные и функции, определяют смысл других инструкций программы.

var

Инструкция var объявляет одну или несколько переменных. Ниже приведен ее синтаксис.

```
var имя_1[=значение_1][,..., имя_n[=значение_n]]
```

После ключевого слова var должен находиться список объявляемых переменных, разделенных запятыми. Каждая переменная может иметь выражение инициализации, задающее начальное значение. Ниже приведен ряд примеров объявления переменных.

```
var i;       // Одна простая переменная
var j = 0;   // Инициализированная переменная
var p, q;    // Две переменные
var greeting = "Привет, " + name;    // Инициализация
var x = 2, y = x*x;                  // В y используется x
var x = 2,                           // Много переменных,
    f = function(x) { return x*x },
    y = f(x);                        // каждая в своей строке
```

Если инструкция var находится в теле функции, она определяет локальную переменную, область видимости которой ограничена телом функции. Если var находится на верхнем уровне кода, она объявляет глобальные переменные, видимые во всей программе JavaScript.

Если в инструкции var не задана инициализация переменной, она при объявлении получает значение undefined.

Инструкция var может быть частью цикла for или for/in.

```
for(var i = 0; i < 10; i++) console.log(i);
for(var i = 0, j=10; i < 10; i++,j--)
```

```
    console.log(i*j);
for(var i in o) console.log(i);
```

function

Ключевое слово `function` используется для определения функции. В главе 3 вы уже встречались с выражением определения функции. Но определение можно написать также в форме инструкции. Рассмотрим два следующих определения функции `f()`.

```
// Выражение определения присваивается переменной
var f = function(x) { return x+1; }
// Инструкция объявления функции
function f(x) { return x+1; }
```

Инструкция объявления функции имеет следующий синтаксис.

```
function имя_функции([аргумент1
                      [, аргумент2
                    [..., аргумент_n]]]) {
   инструкции
}
```

Имя функции должно быть правильным идентификатором. После него в объявлении должен идти список имен параметров функции в скобках. Идентификаторы имен параметров считаются локальными переменными. Их можно использовать в теле функции для ссылки на значения аргументов, передаваемых функции при ее вызове.

Тело функции состоит из произвольного количества инструкций JavaScript и должно быть заключено в фигурные скобки. При определении функции входящие в ее тело инструкции не выполняются. Они только ассоциируются с новым объектом функции, что позволяет им выполняться при ее вызове.

Ниже приведены два примера объявления функций.

```
function hypotenuse(x, y) {
  return Math.sqrt(x*x + y*y);
}

function factorial(n) { // Рекурсивная функция
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

Функция может содержать вызов самой себя, как в последнем примере. В таком случае она называется *рекурсивной*.

Инструкция объявления функции может находиться на верхнем уровне кода JavaScript или в теле другой функции, однако в этом случае она должна находиться на верхнем уровне функции, в которую вложена. Это означает, что инструкция объявления функции не должна находиться внутри инструкции `if`, цикла `while` и т.д.

Инструкция объявления функции отличается от выражения объявления функции тем, что содержит имя функции. В обоих случаях создается объект функции, но инструкция объявляет также имя функции как переменную и присваивает ей объект функции. Как и переменные, объявленные с ключевым словом `var`, инструкции объявления функций неявно "поднимаются" в верхнюю часть сценария или функции, в результате чего все функции сценария или вложенные функции считаются объявленными до начала выполнения кода. Это означает, что функцию можно вызывать перед ее объявлением.

Условия

Условная инструкция выполняет или пропускает другие инструкции в зависимости от значения заданного выражения. Условные инструкции являются как бы точками принятия решения о том, как дальше будет выполняться

программа, по какой ветви пойдет процесс выполнения. Если представить себе, что интерпретатор движется по некоторому маршруту в коде, то условная инструкция представляет собой развилку, на которой интерпретатор может пойти по одному или другому пути.

if

Инструкция `if` — это фундаментальная управляющая инструкция, позволяющая выполнять другие инструкции в зависимости от некоторого условия. Она используется в двух формах. Ниже приведен синтаксис первой формы инструкции `if`.

```
if (выражение) инструкция
```

В этой форме сначала вычисляется выражение. Если его значение преобразуется в `true`, выполняется инструкция. В противном случае инструкция не выполняется, а интерпретатор переходит к инструкции, расположенной непосредственно после блока `if`. Рассмотрим пример использования первой формы инструкции `if`. Если имя пользователя `username` равно `null` или `undefined`, переменной присваивается определенное имя. В противном случае имя остается прежним.

```
if (username == null) username = "Джон Добс";
```

Не забывайте, что выражение обязательно должно быть заключено в скобки.

Во второй форме инструкции `if` используется блок `else`, который выполняется, когда выражение преобразуется в `false`. Ниже приведен синтаксис второй формы.

```
if (выражение)
      инструкция_1
else
      инструкция_2
```

В этой форме первая инструкция выполняется, если выражение преобразуется в `true`, а вторая — в противном случае. В приведенном ниже примере запись в журнал зависит от того, равна ли переменная `n` единице.

```
if (n == 1) {
  console.log("Поступило одно сообщение.");
}
else {
  console.log("Поступило   + n +   сообщений.");
}
```

else if

Инструкция `if/else` вычисляет выражение и в зависимости от результата выполняет одну или другую часть кода. Но что если нужно в зависимости от нескольких условий выполнить или не выполнить много частей кода? Один из способов сделать это состоит в применении инструкции `else if`. Строго говоря, `else if` не является инструкцией языка, это всего лишь часто используемый оборот, образуемый несколькими инструкциями `if/else`.

```
if (n == 1) {
  // Выполнение блока 1
}
else if (n == 2) {
  // Выполнение блока 2
}
else if (n == 3) {
  // Выполнение блока 3
}
else {
  // Выполнение блока 4
}
```

Приведенный выше код — всего лишь последовательность инструкций `if`. Каждая следующая `if` представляет

собой инструкцию очередного блока else, являющегося частью предыдущей инструкции if. Запись else if более удобная, чем ее развернутый эквивалент, приведенный ниже.

```
if (n == 1) {
  // Выполнение блока 1
}
else {
  if (n == 2) {
    // Выполнение блока 2
  }
  else {
    if (n == 3) {
      // Выполнение блока 3
    }
    else {
      // Выполнение блока 4
    }
  }
}
```

switch

Инструкция switch задает ветвление программы. Этого же результата можно достичь с помощью многих инструкций else if, как было показано выше, но если все ветви зависят от значения одного выражения, такое решение будет не самым эффективным, поскольку оно вынуждает многократно вычислять одно и то же выражение. В таком случае лучше применить инструкцию switch.

После ключевого слова switch должны находиться выражение в скобках и блок кода в фигурных скобках.

```
switch(выражение) {
          инструкции

}
```

Однако полный синтаксис инструкции switch более сложный. Разные места блока кода помечены ключевыми словами case, после каждого из которых находятся выражение и двоеточие. Инструкция case напоминает обычную инструкцию с меткой: ей также передается управление. Однако инструкция case помечена не меткой, а выражением, которое сравнивается с выражением инструкции switch. При выполнении инструкции switch интерпретатор сначала вычисляет выражение в скобках (его часто называют *управляющим выражением*), а затем сравнивает его значение с выражениями case с помощью оператора строгого равенства ===. Если интерпретатор находит ключевое слово case со значением, строго равным значению управляющего выражения, он передает управление инструкции, расположенной непосредственно после двоеточия. Если же подходящего ключевого слова case нет, управление передается инструкции, отмеченной меткой default:. Эта метка не обязательна. Если ее нет, то происходит выход из блока switch и управление передается инструкции, расположенной непосредственно после него.

Следующая инструкция switch эквивалентна приведенной выше инструкции if/else.

```
switch(n) {
  case 1:  // Если n === 1
    // Выполнение блока 1
    break; // Прекратить выполнение
  case 2:  // Если n === 2
    // Выполнение блока 2
    break; // Прекратить выполнение
  case 3:  // Если n === 3
    // Выполнение блока 3
    break; // Прекратить выполнение
  default: // Если ни один case не подходит
    // Выполнение блока 4
    break;
}
```

Обратите внимание на ключевое слово `break` после каждой инструкции `case` в приведенном выше коде. Инструкция `break` рассматривается далее. В данном коде она передает управление инструкции, расположенной после блока `switch`, в результате чего выполнение инструкции `switch` немедленно завершается. Таким образом, ключевое слово `case` задает только начальную точку, но не определяет конечную точку выполнения предыдущего блока `case`. Если в этой точке опустить инструкцию `break`, выполнение блока `switch` будет продолжено; интерпретатор выйдет из блока `switch`, только когда дойдет до его конца или встретит инструкцию `break`. Очевидно, это не то, что нужно для решения задачи выбора. Поэтому каждый блок `case` необходимо заканчивать инструкцией `break` или `return`.

Ниже приведен более реалистичный пример использования инструкции `switch`. Функция `convert()` преобразует значение `x` в строку, причем способ преобразования зависит от типа значения `x`.

```
function convert(x) {
  switch(typeof x) {
    case 'number': // Преобразование в
                   // шестнадцатеричное целое
      return x.toString(16);
    case 'string': // Заключение в кавычки
      return '"' + x + '"';
    default:       // Любой другой тип
      return String(x);
  }
}
```

В предыдущих двух примерах после ключевых слов `case` находятся числовые или строковые литералы. Это наиболее распространенный способ использования инструкций `switch`, однако спецификация ECMAScript позволяет использовать в блоке `case` произвольное выражение.

Циклы

Инструкция цикла заставляет интерпретатор многократно выполнять один и тот же блок кода, который называется *телом цикла*. Каждое однократное выполнение тела цикла называется *итерацией*. В JavaScript доступны четыре типа циклов: `while`, `do/while`, `for` и `for/in`.

while

Инструкция `while` определяет простейший цикл, имеющий следующий синтаксис.

```
while (выражение_цикла)
        тело_цикла
```

Дойдя до инструкции `while`, интерпретатор в первую очередь вычисляет выражение в скобках. Если оно преобразуется в `false`, интерпретатор пропускает тело цикла и продолжает выполнение программы. Если же оно преобразуется в `true`, интерпретатор выполняет тело цикла и опять вычисляет выражение цикла. Этот процесс повторяется до тех пор, пока на очередной итерации выражение не вернет `false`. Если оно никогда не вернет `false`, цикл будет выполняться бесконечно. Обратите внимание на то, что для умышленного создания бесконечного цикла достаточно записать `while(true)`.

Ниже приведен пример цикла `while`, который выводит на консоль числа от 0 до 9.

```
var count = 0;
while (count < 10) {
  console.log(count);
  count++;
}
```

Сначала переменная count (она называется *счетчиком цикла*) имеет значение 0. На каждой итерации цикла она увеличивается на единицу. Когда она становится равной 10, выражение возвращает false, в результате чего выполнение инструкции while заканчивается и управление передается инструкции, расположенной непосредственно после закрывающей фигурной скобки.

do/while

Цикл do/while отличается от while тем, что выражение цикла вычисляется не в начале, а в конце тела цикла. Следовательно, тело цикла в любом случае выполняется хотя бы один раз. Ниже приведен синтаксис цикла do/while.

```
do
     инструкция
while(выражение_цикла);
```

В приведенном ниже примере цикл do/while выполняется до тех пор, пока счетчик i не станет равным длине массива.

```
function printArray(a) {
  var len = a.length, i = 0;
  if (len == 0)
    console.log("Пустой массив");
  else {
    do {
      console.log(a[i]);
    } while (++i < len);
  }
}
```

for

Инструкция for упрощает кодирование циклов, соответствующих определенному шаблону. В большинстве циклов присутствует переменная счетчика в том или ином

виде. Эту переменную нужно инициализировать перед первой итераций, обновлять перед или после каждой итерации, а также проверять ее значение перед каждой итерацией. В цикле такого вида инициализация, проверка и обновление значения счетчика – три стандартные операции. В синтаксисе цикла `for` каждой из этих операций посвящено отдельное выражение, что делает структуру цикла довольно наглядной.

```
for(инициализация; проверка; обновление)
    инструкция
```

Выражения инициализации, проверки и обновления счетчика отделяются друг от друга точками с запятыми. Они помещены в первую строку инструкции цикла, поэтому хорошо видны, и можно легко понять, что они делают. Если программист случайно забудет закодировать одну из этих операций, ошибка будет видна с первого взгляда.

Чтобы объяснить, как работает цикл `for`, рассмотрим эквивалентный цикл `while`.

```
инициализация;
   while(проверка) {
      инструкция
      обновление;
   }
```

Выражение инициализации вычисляется один раз перед началом работы цикла. Чтобы выражение инициализации было полезным, оно обязательно должно иметь побочные эффекты (обычно таким эффектом служит операция присваивания). В JavaScript можно объявить и инициализировать переменную счетчика с помощью инструкции `var` непосредственно в заголовке цикла. Областью действия этой переменной будет тело цикла. Выражение *проверка* вычисляется перед каждой итерацией. От его значения зависит, будет ли выполняться тело

цикла в очередной раз или произойдет выход из цикла. Если значение выражения проверки можно преобразовать в `true`, выполняется тело цикла. В конце тела цикла выполняется выражение *обновление*. Чтобы оно было полезным, оно, как и выражение инициализации, должно иметь побочные эффекты. В общем случае таким побочным эффектом служит операция присваивания, инкремента или декремента.

Приведенный ниже код выводит на консоль числа от 0 до 9 с помощью цикла `for`. Сравните его с эквивалентным примером на основе цикла `while`, приведенным выше.

```
for(var count = 0; count < 10; count++)
   console.log(count);
```

for/in

В инструкции `for/in` используется ключевое слово `for`, но на этом сходство с циклом `for` заканчивается. Этот цикл работает совершенно иначе. Ниже приведен его синтаксис.

```
for (переменная in объект)
    инструкция
```

В качестве переменной можно подставить имя переменной или инструкцию `var`, объявляющую одну переменную. Выражение *объект* должно возвращать объект определенного типа. Тело цикла, как и в других циклах, может быть инструкцией или блоком инструкций (важно отметить, что блок также считается инструкцией).

Обычный цикл `for` легко применить для обхода элементов массива.

```
for(var i = 0; i < a.length; i++)
console.log(a[i]); // Вывод каждого элемента
```

Цикл `for/in` позволяет так же легко сделать это для свойств объекта. Фактически он перечисляет имена свойств объекта.

```
for(var p in o)
  console.log(o[p]); // Вывод каждого свойства
```

Подойдя к циклу `for/in`, интерпретатор в первую очередь вычисляет выражение *объект*, а затем выполняет тело цикла по одному разу для каждого перечислимого свойства этого объекта. Перед каждой итерацией интерпретатор присваивает имя свойства переменной цикла.

В цикле `for/in` перечисляются не все свойства объекта, а только *перечислимые* (подробнее об этом — в главе 5). Встроенные методы, определенные в базовой библиотеке JavaScript, не являются перечислимыми. Например, все встроенные объекты имеют метод `toString()`, но цикл `for/in` не считает его перечислимым свойством. Все свойства и методы, определенные в коде, являются перечислимыми. Однако, согласно ECMAScript 5, их можно сделать неперечислимыми, как показано в главе 5.

Спецификация ECMAScript не определяет порядок перечисления свойств объекта в цикле `for/in`. Однако на практике интерпретаторы JavaScript, реализованные в браузерах всех известных производителей, перечисляют свойства простых объектов в том же порядке, в каком они были определены: более старые свойства идут первыми. Если объект был создан как объектный литерал, последовательность перечисления совпадает с последовательностью записи свойств в литерале. Обратите внимание на то, что данное правило применимо не ко всем объектам. В частности, если объект содержит индексированные свойства массива, эти свойства перечисляются в числовой последовательности, а не в порядке их создания.

Переходы

Инструкция перехода вынуждает интерпретатор JavaScript перейти в заданное место исходного кода. Инструкция `break` задает выход из цикла или другого блока. Инструкция `continue` задает пропуск всей оставшейся части тела цикла и переход к началу новой итерации. В JavaScript можно отмечать инструкции метками, что позволяет задать в строке `break` или `continue` целевую инструкцию. Инструкция `return` вынуждает интерпретатор выйти из текущей функции, вернуть в вызывающую функцию вычисленное значение и перейти к следующей инструкции в вызывающей функции. Инструкция `throw` генерирует исключение, приводящее к переходу в другое место кода. Обычно она используется совместно с блоками `try/catch/finally`, которые определяют способ обработки исключения.

Помеченные инструкции

Любую инструкцию можно пометить, записав перед ней идентификатор метки и двоеточие.

идентификатор: *инструкция*

Пометив инструкцию, вы даете ей имя, на которое можно ссылаться в других местах программы. Пометить можно любую инструкцию, но польза от этого будет только для инструкции с телом, например для цикла или условной инструкции. Присвоив циклу имя, можно использовать инструкции `break` и `continue` в теле цикла для выхода из него или перехода непосредственно в верхнюю часть цикла и выполнения следующей итерации. В JavaScript `break` и `continue` — единственные инструкции, позволяющие использовать метки (подробнее об этом — в следую-

щем разделе). Ниже приведен пример помеченного цикла while. Метка используется инструкцией continue.

```
mainloop: while(token != null) {
  // Код цикла
  continue mainloop; // Переход к началу цикла
  // Код цикла
}
```

break

Инструкция break без метки приводит к немедленному выходу из внутреннего цикла или блока switch. Ее синтаксис очень простой.

```
break;
```

Поскольку в такой форме инструкция задает действие в теле цикла или блоке switch, она может находиться только в этих двух местах.

Выше вы уже видели примеры использования инструкции break в блоке switch. В циклах она используется главным образом для преждевременного выхода, когда по какой-либо причине продолжать выполнение цикла не имеет смысла. Кроме того, когда цикл имеет сложное условие завершения, часто легче выйти из него с помощью инструкции break, чем пытаться выразить сложное условие в единственном выражении цикла. Приведенный ниже код ищет элемент массива, имеющий значение target. Цикл завершается обычным способом, если такого элемента нет. Если же он есть, инструкция break завершает выполнение цикла.

```
for(var i = 0; i < a.length; i++) {
  if (a[i] == target) break;
}
```

Хотя это и редко используется на практике, JavaScript позволяет записать имя метки после ключевого слова `break` (только имя, без двоеточия).

```
break имя_метки;
```

Если инструкция `break` записана с меткой, она завершает цикл или блок инструкции, помеченной данной меткой. Если внешнего блока с данной меткой нет, будет сгенерирована ошибка. Именованная инструкция необязательно должна быть циклом или блоком `switch`. С помощью инструкции `break` с меткой можно выйти из любого помеченного блока.

continue

В отличие от инструкции `break`, которая завершает цикл, инструкция `continue` передает управление следующей итерации цикла. Синтаксис инструкции `continue` столь же прост.

```
continue;
```

Эту инструкцию также можно записать с именем метки.

```
continue имя_метки;
```

В обеих формах — с меткой и без — эту инструкцию можно использовать только в теле цикла. В любом другом месте она вызовет синтаксическую ошибку.

В приведенном ниже примере инструкция `continue` используется для прекращения текущей и запуска новой итерации, если элемент массива не является числом (в противном случае произошла бы ошибка).

```
for(i = 0; i < data.length; i++) {
  if (isNaN(data[i])) continue;
  total += data[i];
}
```

Инструкция continue с меткой обычно используется во вложенных циклах, когда нужно задать, в какой цикл нужно перейти.

return

Выше уже упоминалось о том, что вызов функции является выражением, и, как любое другое выражение, он имеет некоторое значение. Инструкция return, записанная в теле функции, определяет значение, возвращаемое функцией, т.е. значение выражения вызова. Ниже приведен синтаксис инструкции return.

```
return выражение;
```

Инструкция return может находиться только в теле функции. В любом другом месте она приведет к синтаксической ошибке. Когда интерпретатор встречает инструкцию return, он прекращает выполнение функции, вычисляет выражение и возвращает значение выражения в вызывающую функцию. В вызывающей функции оно становится значением выражения вызова. В приведенном ниже примере переменная y получает значение 4.

```
function square(x) { return x*x; }
y = square(2);
```

Если в теле функции нет инструкции return, функция выполняется до конца (т.е. до последней закрывающей фигурной скобки) и возвращает значение undefined. Инструкцию return можно записать и без выражения; встретив ее, интерпретатор немедленно прекратит выполнение тела функции, а функция вернет значение undefined. В приведенном ниже примере инструкция return завершает выполнение функции, если полученный объект неопределенный или его не существует (значение переменной объекта равно null или undefined).

```
function display_object(o) {
  if (!o) return;
  // Остальные инструкции в теле функции
}
```

throw

Исключение— это сигнал о том, что во время выполнения кода произошла ошибка. Обычно ошибку генерирует интерпретатор, однако программист может принудительно задать генерацию ошибки в определенном месте, вставив в код инструкцию `throw`. Как и обычную ошибку, ее можно перехватить и обработать с помощью инструкции `catch`.

Инструкция `throw` имеет следующий синтаксис.

```
throw выражение;
```

Выражение может возвращать значение любого типа. Обычно задают возврат числа, служащего номером, или кодом, ошибки. Иногда задают возврат строки, содержащей сообщение об ошибке. Класс `Error` и его подклассы используются, когда ошибки генерируются интерпретатором и их можно перехватить и обработать. Объект `Error` имеет свойство `name`, указывающее на тип ошибки, и свойство `message`, содержащее строку с объяснением ошибки, которую можно отобразить для пользователя. Ниже приведен пример функции вычисления факториала, которая при получении неправильного аргумента генерирует объект `Error`.

```
function factorial(x) {
  // Если x неправильное, генерируется исключение
  if (x < 0) throw new Error("x должно быть положительным");
  // Вычисление возвращаемого значения
  for(var f = 1; x > 1; f *= x, x--);
  return f;
}
```

Когда возникает исключение, интерпретатор JavaScript немедленно прекращает нормальное выполнение программы и переходит к ближайшему обработчику исключений. Обработчик находится в блоке `catch` инструкции `try/catch/finally`, которая рассматривается в следующем разделе. Если с блоком, в котором возникло исключение, не ассоциирован ни один блок `catch`, интерпретатор ищет обработчик во внешнем блоке кода и продвигается вверх по уровням вложенности блоков, пока не найдет обработчик. Если исключение сгенерировано в функции, в которой нет инструкции `try/catch/finally`, то интерпретатор, естественно, не найдет обработчик в данной функции. В этом случае он ищет обработчик в вызывающих функциях, продвигаясь вверх по стеку вызова. Если же обработчика нет и выше по стеку, исключение считается ошибкой. Тогда выполнение программы прекращается, и пользователь получает сообщение об ошибке.

try/catch/finally

Инструкция `try/catch/finally` реализует встроенный в JavaScript механизм обработки исключений. Блок `try` содержит инструкции, исключения которых обрабатываются блоком `catch`. Иными словами, блок `catch` выполняется, если в блоке `try` возникло исключение. После блока `catch` может находиться блок `finally`, содержащий код очистки, который гарантированно будет выполнен независимо от того, что происходит в блоке `try`. Блоки `catch` и `finally` необязательные, но после каждого блока `try` должен быть как минимум один из них. Все блоки `try`, `catch` и `finally` заключены в фигурные скобки, которые нельзя опустить, даже если блок состоит из одной инструкции.

Приведенный ниже код иллюстрирует синтаксис и назначение инструкций `try/catch/finally`.

```
try {
  // Обычно расположенный здесь код выполняется
  // с начала до конца, но при возникновении
  // исключения его выполнение прекращается и
  // управление передается блоку catch.
  // Исключение может возникнуть в инструкции или
  // в теле вызванной функции; кроме того,
  // исключение может быть принудительно
  // сгенерировано с помощью инструкции throw.
}
catch(e) {
  // Инструкции этого блока выполняются, только
  // если блок try сгенерировал исключение.
  // Локальную переменную e можно использовать
  // для ссылки на объект Error или другое
  // сгенерированное значение. Блок catch может
  // обработать исключение, проигнорировать его
  // или повторно сгенерировать сигнальное
  // исключение с помощью инструкции throw.
}
finally {
  // Этот блок выполняется всегда независимо от
  // того, что произошло в блоке try, т.е. в
  // следующих случаях:
  // 1)выполнение блока try нормально завершено;
  // 2)выполнение блока try прервано инструкцией
  //   break, continue или return;
  // 3)исключение обработано блоком catch;
  // 4)необработанное исключение передано
  //   вверх по стеку вызовов.
}
```

Обратите внимание на то, что после ключевого слова `catch` находится идентификатор в скобках, что очень напоминает вызов функции с одним параметром. Когда возникает исключение, этому параметру присваивается некоторое значение, ассоциированное с исключением, чаще всего — объект ошибки `Error`. В отличие от обычных переменных, этот идентификатор имеет область видимо-

сти, совпадающую с блоком `catch`, т.е. его можно использовать только в блоке `catch`.

Ниже приведен реалистичный пример использования инструкции `try/catch`. В нем используется метод `factorial()`, упомянутый в предыдущем разделе и вычисляющий факториал заданного числа. Для ввода данных и вывода результата используются встроенные в JavaScript методы `prompt()` и `alert()`. Если введенное число отрицательное или введенную строку нельзя преобразовать в число, в функции `factorial()` будет сгенерировано исключение.

```
try {
  // Приглашение ввести число
  var n = Number(prompt("Введите число", ""));
  // Вычисление факториала исходя из предположения,
  // что введено правильное число
  var f = factorial(n);
  // Вывод результата
  alert(n + "! = " + f);
}
catch (ex) { // Произошло исключение
  alert(ex); // Вывод сообщения об ошибке
}
```

Другие инструкции

В этом разделе рассматриваются инструкции `with`, `debugger` и `use strict`, не относящиеся к упомянутым выше категории.

with

Интерпретатор ищет значение переменной сначала среди переменных, определенных в текущей функции, затем (если функция вложенная) — во внешней функции и наконец — среди глобальных переменных. Инструкция `with`

временно изменяет данную последовательность поиска переменной. Она задает объект, свойства которого должны интерпретироваться как переменные. Ниже приведен синтаксис инструкции `with`.

```
with (объект)
    инструкция
```

Инструкция может быть (и чаще всего является) блоком, который выполняется так, как будто он служит телом вложенной функции, причем свойства объекта переданы этой функции через список параметров.

В строгом режиме (см. следующий раздел) инструкция `with` запрещена. В нестрогом режиме рекомендуется считать ее устаревшей и применять как можно реже, только если в этом есть крайняя необходимость. Интерпретатору тяжело оптимизировать код, в котором есть инструкция `with`, поэтому такой код выполняется намного медленнее, чем эквивалентный код без инструкции `with`.

debugger

Инструкция `debugger` в нормальном режиме ничего не делает. Однако если программа отладчика доступна и выполняется, то рабочая среда может (хотя и необязательно, в зависимости от реализации рабочей среды) выполнить некоторые действия, полезные для отладки программы с инструкцией `debugger`. Чаще всего инструкция `debugger` работает как точка прерывания: выполнение кода JavaScript в этой точке приостанавливается, и программист может с помощью отладчика просмотреть значения переменных, состояние стека и другие параметры выполнения. Предположим, что в функции `f()` генерируется исключение по той причине, что она вызывается с неопределенным аргументом, и вы не можете понять, с какого места вызвана функция. Без отладчика выяснить

это непросто, потому что программа может быть большой и содержать много точек вызова этой функции. Для решения данной проблемы измените функцию `f()`. Она должна начинаться следующим образом.

```
function f(o) {
  if (o===undefined) debugger; // Точка прерывания
  // Остальная часть тела функции
}
```

Теперь при вызове функции `f()` без аргумента или с неопределенным аргументом выполнение будет остановлено, и можно будет применить отладчик для просмотра стека вызовов и поиска неправильного выражения вызова.

Формально инструкция `debugger` была добавлена в язык только спецификацией ECMAScript 5, однако в реальности она уже давно реализована во многих браузерах.

"use strict"

Директива `"use strict"` введена в ECMAScript 5 для задания строгого режима выполнения кода. Директива — это не инструкция, но достаточно близкое понятие, чтобы можно было рассмотреть ее в разделе, посвященном инструкциям. Во фразе `"use strict"` нет ключевых слов JavaScript, это лишь строковый литерал, который игнорируется более ранними интерпретаторами ECMAScript 3. Если она помещена в начала сценария или тела функции, она имеет специальное значение для интерпретатора ECMAScript 5.

Назначение директивы `"use strict"` состоит в том, чтобы при выполнении данного кода переключить интерпретатор в *строгий режим*. В строгом режиме используется ограниченное подмножество языка, устраняющее ряд существенных дефектов платформы и обеспечивающее

повышенные уровни проверки ошибок и безопасности. Ниже рассматриваются наиболее важные различия между строгим и нестрогим режимами.

- Инструкция `with` в строгом режиме запрещена.
- В строгом режиме все переменные должны быть объявлены. Если присвоить значение идентификатору, не являющемуся объявленной переменной, параметром или свойством глобального объекта, то будет сгенерирована ошибка `ReferenceError`. В нестрогом режиме в этом случае будет неявно создана глобальная переменная путем добавления нового свойства в глобальный объект.
- В строгом режиме в функции, вызванной как функция (а не метод объекта), ключевое слово `this` имеет значение `undefined`. В нестрогом режиме всем функциям, вызываемым как функции, посредством значения `this` передается глобальный объект. Это отличие можно использовать для выяснения того, в каком режиме работает код.

```
var hasStrictMode = (function() {
   "use strict";
   return this === undefined;
}());
```

- В строгом режиме присвоение значения свойству, созданному в режиме "только чтение", а также попытка создания свойства несуществующего объекта приводят к генерированию ошибки `TypeError`. В нестрогом режиме интерпретатор никак не реагирует на неудачу. Аналогичным образом в строгом режиме попытка удалить неконфигурируемое свойство или значение приводит к ошибке `TypeError`

или SyntaxError. В нестрогом режиме попытка приводит к неудаче (но интерпретатор никак не реагирует на это), и выражение delete возвращает значение false.

- В строгом режиме код, передаваемый функции eval(), не может объявлять переменные или определять функции в области видимости вызывающей функции. Переменные и функции в этом случае находятся в новой области видимости, созданной функцией eval(). При завершении функции eval() эта область видимости уничтожается.
- В строгом режиме не разрешены восьмеричные целые литералы (начинающиеся с 0 без следующего символа x). Впрочем, в нестрогом режиме восьмеричные литералы поддерживаются не во всех реализациях рабочей среды.
- В строгом режиме идентификаторы eval и arguments считаются ключевыми словами, и присваивать им значения запрещено.

Объекты

Объект — фундаментальный тип данных JavaScript. Объект представляет собой составное значение: он состоит из многих значений (примитивных значений и других объектов) и позволяет сохранять и извлекать эти значения и объекты по именам. Объект содержит неупорядоченную коллекцию *свойств*, каждое из которых содержит имя и значение. Имя свойства является строкой, поэтому можно считать, что объекты связывают строки со значениями. Привязка строк к значениям представляет собой структуру данных, которая может называться по-разному: "хеш", "хеш-таблица", "словарь" или "ассоциативный массив". Но объект — это не только набор пар "имя–значение". Кроме собственного набора свойств, объект JavaScript может наследовать свойства другого объекта, который называется *прототипом*. Методы объекта обычно являются унаследованными свойствами, а наследование прототипов — ключевая особенность JavaScript.

Объекты JavaScript являются динамическими. Это означает, что свойства обычно можно добавлять и удалять. Впрочем, в JavaScript ничто не мешает имитировать статические объекты или структуры, используемые в статически типизированных языках программирования. Кроме того, объекты можно использовать для представления наборов строк (игнорируя ассоциированные с ними значения).

Любое значение JavaScript, не являющееся строкой, числом или значением `true`, `false`, `null` или `undefined`, является объектом.

Объекты можно изменять, манипулируя ими посредством ссылок на них. Например, если переменная x ссылается на объект и в коде выполняется инструкция var y=x;, то данная инструкция присваивает переменной y ссылку на этот же объект, а не на его копию. Любые изменения, сделанные через переменную y, будут видны через переменную x.

Создание объектов

Объекты можно создавать с помощью объектных литералов, ключевого слова new и функции Object.create().

Объектные литералы

Проще всего создать объект, включив объектный литерал в код. *Объектный литерал* — это список разделенных запятой пар "имя–значение", заключенный в фигурные скобки. Имя свойства и значение отделены друг от друга двоеточием. Имя свойства представляет собой идентификатор или строковый литерал (допустима пустая строка). Значением свойства может быть любое выражение JavaScript. Значение выражения (это может быть примитивное или объектное значение) становится значением свойства. Ниже приведен ряд примеров создания объектов с помощью объектных литералов.

```
var empty = {};                 // Объект без свойств
var point = { x:0, y:0 };  // Два свойства
var point2 = {                  // Объектный литерал с более
x:point.x,                      // сложными свойствами
y:point.y+1
};
var book = { // Имена свойств, не являющиеся
             // идентификаторами, должны быть
             // заключены в двойные кавычки
// В именах-литералах разрешены пробелы, ключевые
```

```
// слова и знаки пунктуации
"main title": "JavaScript",
'sub-title': "Pocket Ref",
"for": "all audiences",
};
```

Ключевое слово new

Оператор new создает и инициализирует новый объект. После ключевого слова new должен находиться вызов функции. Используемая таким образом функция называется *конструктором* и служит для инициализации создаваемого объекта. Базовая библиотека JavaScript содержит встроенные конструкторы всех встроенных типов. Ниже приведен ряд примеров создания встроенных типов.

```
// Пустой объект; то же, что и {}
var o = new Object();
// Пустой массив; то же, что и []
var a = new Array();
// Объект даты, содержащий текущее время
var d = new Date();
// Объект регулярного выражения
var r = new RegExp("js");
```

Кроме встроенных конструкторов, программист может создавать и применять конструкторы для инициализации любых создаваемых объектов. Подробнее этот вопрос рассматривается в главе 8.

Прототипы

Прежде чем рассмотреть третью методику создания объектов, необходимо понять, что такое прототипы. С каждым объектом JavaScript ассоциирован другой объект JavaScript (иногда null, но редко). Этот второй объект называется *прототипом*, а первый объект наследует свойства прототипа.

Все объекты, созданные объектными литералами, имеют один и тот же прототип, на который можно ссылаться как на `Object.prototype`. Объекты, созданные с помощью ключевого слова `new` и конструктора, имеют в качестве своего прототипа значение свойства `prototype`. Например, объект `new Object()` имеет прототип `Object.prototype`, как и объект, созданный с помощью литерала `{}`. Аналогично объект `new Array()` имеет прототип `Array.prototype`, а объект `new.Date()` — прототип `Date.prototype`.

Объект `Object.prototype` — один из редких объектов, не имеющих прототипа и не наследующих никаких свойств. Другие объекты прототипов — это обычные объекты, имеющие прототипы. Все встроенные и большинство пользовательских конструкторов имеют прототипы, наследуемые от `Object.prototype`. Например, объект `Date.prototype` наследует свойства объекта `Object.prototype`, поэтому объект `Date`, созданный с помощью выражения `new Date()`, наследует свойства как `Date.prototype`, так и `Object.prototype`. Этот связанный ряд объектов прототипов называется *цепочкой прототипов*.

Наследование свойств и получение прототипа объекта рассматриваются в следующих разделах. Связь между прототипами и конструкторами подробнее рассматривается в главе 8. Там же вы узнаете о том, как определить новые "классы" объектов с помощью функций конструкторов и путем присвоения объекта прототипа свойству `prototype`.

Функция `Object.create()`

В ECMAScript 5 определена функция `Object.create()`, которая создает объект, используя первый аргумент в качестве прототипа объекта. Кроме того, функция `Object.create()` принимает необязательный второй аргумент,

описывающий свойства нового объекта. Подробнее второй аргумент рассматривается в следующих разделах.

`Object.create()` — это статическая функция, а не метод, вызываемый через индивидуальные объекты. Ей нужно передать прототип объекта.

```
// Объект o1 наследует свойства x и y
var o1 = Object.create({x:1, y:2});
```

Можно передать значение `null`, но тогда новый объект не будет иметь прототипа и не унаследует ничего, даже базового метода `toString()`, а это означает, что объект не будет работать с оператором +.

```
// Объект o2 не наследует ни свойств, ни методов
var o2 = Object.create(null);
```

Если нужно создать обычный пустой объект (такой, как возвращаемый выражением `{}` или `new Object()`), передайте функции `Object.create()` прототип `Object.prototype`.

```
// Объект o3 аналогичен {} или new Object()
var o3 = Object.create(Object.prototype);
```

Возможность создать объект с произвольным прототипом (иными словами, создать "наследник" любого объекта) — мощное средство ECMAScript 5. В листинге 5.1 показано, как его можно имитировать средствами ECMAScript 3. Как видите, код довольно длинный и громоздкий.

Листинг 5.1. Создание объекта, наследующего прототип

```
// Функция inherit() возвращает объект, наследующий
// свойства прототипа p. В нем используется
// введенная в ECMAScript 5 функция Object.create(),
// если она определена. В противном случае
// применяется устаревшая методика
```

```
function inherit(p) {
  if (p == null)       // Объект p не должен быть равен null
    throw TypeError();
  if (Object.create)   // Вызов Object.create()
    return Object.create(p);
  var t = typeof p;    // Убедимся, что p -- объект
  if (t !== "object" && t !== "function")
    throw TypeError();
  function f() {}; // Определение конструктора
  f.prototype = p; // Установка свойства prototype
  return new f();  // Создание наследника p
}
```

Код тела функции `inherit()` будет более понятным после ознакомления с концепцией конструкторов в главе 8.

Свойства

Наиболее важная часть объекта — его свойства.

Чтение и запись свойств

Получить значение свойства можно с помощью квадратных скобок [] или оператора "точка", как описано в главе 3. Слева от точки или квадратных скобок должно быть выражение, значением которого является объект. При использовании точки справа от нее должен быть идентификатор, обозначающий имя свойства. При использовании квадратных скобок в них должно быть выражение, возвращающее число или строку с именем свойства.

```
// Чтение свойства author объекта book
var author = book.author;
// Чтение свойства surname объекта author
var name = author.surname
// Чтение свойства "main title" объекта book
var title = book["main title"]
```

Для создания или записи свойства используется точка или квадратные скобки, как и для чтения, но они должны находиться слева от оператора присваивания.

```
// Создание свойства edition объекта book
book.edition = 6;
// Запись свойства "main title" объекта book
book["main title"] = "ECMAScript";
```

Наследование свойств

Объекты JavaScript имеют набор "собственных свойств" и, кроме того, наследуют набор свойств от своих прототипов. Чтобы понять эту концепцию, рассмотрим подробнее способы обращения к свойствам. В примерах данной главы используется функция `inherit()`, приведенная в листинге 5.1, которая создает объекты на основе заданных прототипов.

Предположим, нужно прочитать свойство `x` объекта `o`. Если у объекта `o` нет собственного свойства с именем `x`, выполняется его поиск в прототипе. Если в прототипе нет собственного свойства с этим именем, но есть свой прототип, выполняется поиск свойства в прототипе прототипа. Этот процесс продолжается до тех пор, пока не будет найдено свойство `x` или не будет достигнут прототип, свойство `prototype` которого имеет значение `null`. Таким образом, атрибут прототипа создает цепочку или связанный список прототипов, из которых наследуются свойства (важно отметить, что в JavaScript методы объекта также считаются свойствами).

```
// Объект o наследует методы из Object.prototype
var o = {}
o.x = 1; // Объект имеет собственное свойство x.
// Объект p наследует o из Object.prototype
var p = inherit(o);
p.y = 2; // p имеет собственное свойство y
```

```
// q наследует свойства из p, o и Object.prototype
var q = inherit(p);
q.z = 3; // q имеет собственное свойство z
// Метод toString унаследован от Object.prototype
var s = q.toString();
// x и y унаследованы от o и p
q.x + q.y // => 3
```

Предположим, нужно присвоить некоторое значение свойству x объекта o. Если o имеет собственное (не унаследованное от прототипа) свойство с именем x, то операция присвоения всего лишь изменяет значение существующего свойства. В противном случае оператор присваивания создает еще одно свойство объекта o с именем x. Если объект o перед этим унаследовал свойство x, это унаследованное свойство теперь будет скрыто вновь созданным собственным свойством с этим же именем.

Удаление свойств

Оператор delete (см. главу 3) удаляет свойство из объекта. Единственный операнд этого оператора должен быть выражением доступа к свойству. Как ни странно, оператор delete удаляет не значение свойства, а само свойство.

```
delete book.author; // Теперь свойства author нет
delete book["main title"];
```

Оператор delete удаляет только собственные свойства, но не унаследованные. Чтобы удалить унаследованное свойство, его нужно удалить из объекта прототипа, в котором оно определено. Эта операция затронет все объекты, наследующие (непосредственно или косвенно) данный прототип.

Проверка свойств

Объект JavaScript можно представлять себе, как набор свойств. Часто полезно проверить, что содержится в этом наборе, есть ли в объекте свойство с заданным именем и какое это свойство. Это можно сделать с помощью оператора `in` и методов `hasOwnProperty()` и `propertyIsEnumerable()`. Еще один способ проверки состоит в чтении свойств и анализе полученного результата.

Оператор `in` принимает строку с именем свойства с левой стороны и объект с правой стороны. Если объект имеет собственное или унаследованное свойство с этим именем, оператор `in` возвращает `true`.

```
var o = { x: 1 }
"x" in o;         // true: объект o имеет свойство x
"y" in o;         // false: объект o не имеет свойства y
"toString" in o; // true: унаследованное свойство
```

Принадлежащий объекту метод `hasOwnProperty()` проверяет, имеет ли объект собственное свойство с заданным именем. Если свойство с этим именем унаследованное, метод возвращает `false`.

```
var o = { x: 1 }
o.hasOwnProperty("x"); // true: есть свойство x
o.hasOwnProperty("y"); // false: есть свойство y
// Свойство toString унаследованное
o.hasOwnProperty("toString"); // false
```

Если собственное свойство с заданным именем существует, то с помощью метода `propertyIsEnumerable()` можно проверить, является ли оно перечислимым. Этот метод возвращает `true`, если указанное свойство собственное и атрибут `enumerable` равен `true`. Некоторые встроенные свойства неперечислимые. Свойства, созданные обычным кодом JavaScript, являются перечислимыми,

если только не задано обратное с помощью методов, определенных в спецификации ECMAScript 5 (они рассматриваются в следующих разделах).

```
var o = inherit({ y: 2 });
o.x = 1;
// Объект o имеет перечислимое свойство x
o.propertyIsEnumerable("x"); // true
// Свойство y унаследованное
o.propertyIsEnumerable("y"); // false
// false: метод toString неперечислимый
Object.prototype.propertyIsEnumerable("toString");
```

Вместо использования оператора `in` обычно можно выполнить следующее: сначала прочитать значение свойства, а затем с помощью оператора `!==` проверить, определено ли оно.

```
var o = { x: 1 }
o.x !== undefined;         // true
o.y !== undefined;         // false
o.toString !== undefined; // true
```

Однако есть одна операция, которую можно выполнить только с помощью оператора `in`, но не с помощью приведенного выше способа (чтение и проверка оператором `!==`). Оператор `in` отличает несуществующие свойства от свойств, которые существуют, но имеют значение `undefined`. Рассмотрим следующий код.

```
var o = { x: undefined }
o.x !== undefined // false: свойство не определено
o.y !== undefined // false: свойство не существует
"x" in o          // true: свойство существует
"y" in o          // false: свойство не существует
delete o.x;       // Удаление свойства x
"x" in o          // false: теперь свойства x нет
```

Перечисление свойств

Часто нужно работать не с индивидуальными свойствами, а пройти по всем свойствам объекта автоматически или получить их список. Обычно это делают с помощью цикла `for/in`, но спецификация ECMAScript 5 предоставляет два удобных альтернативных способа.

Цикл `for/in` рассмотрен в главе 4. Тело цикла `for/in` выполняется по одному разу для каждого перечислимого свойства заданного объекта, причем имя свойства присваивается переменной цикла. Унаследованные объектом встроенные методы не являются перечислимыми. Но перечислимыми, кроме прочего, являются свойства, добавленные кодом в объект (если в этом же коде не указано обратное). Рассмотрим пример, в котором объект `o` имеет три перечислимых собственных свойства.

```
var o = {x:1, y:2, z:3};
// Унаследованные методы не перечислимые
o.propertyIsEnumerable("toString") // => false
// Вывод в цикле значений x, y и z
for(p in o) console.log(p);
```

В большинстве библиотек новые методы и свойства добавляются в `Object.prototype` таким образом, что они наследуются и доступны для всех объектов. Однако до ECMAScript 5 не было способа сделать добавляемые методы неперечислимыми, поэтому все они перечислялись в цикле `for/in`. Для решения этой проблемы можно фильтровать свойства, возвращаемые объектом цикла `for/in`. Ниже приведены два способа фильтрации.

```
for(p in o) {
  if(!o.hasOwnProperty(p)) // Пропуск свойства
}

for(p in o) {
```

```
    if(typeof o[p] ==="function") // Пропуск метода
    continue;
}
```

Ниже приведена функция, которая с помощью цикла `for/in` копирует перечислимые свойства из объекта `p` в объект `o` и возвращает ссылку на `o`. Если `o` и `p` имеют свойство с одним и тем же именем, свойство в объекте `o` переопределяется.

```
function extend(o, p) {
  for(prop in p) {     // Все свойства объекта p
    o[prop] = p[prop]; // Добавление свойства в o
  }
  return o;
}
```

Кроме цикла `for/in`, спецификация ECMAScript 5 определяет две функции, которые перечисляют имена свойств. Первая, `Object.keys()`, возвращает массив имен собственных перечислимых свойств объекта. Вторая, `Object.getOwnPropertyNames()`, в отличие от `Object.keys()`, возвращает имена всех собственных свойств заданного объекта, а не только перечислимых.

Сериализация свойств и объектов

Сериализация — это процесс преобразования состояния объекта в строку, из которой позже можно будет восстановить состояние. Спецификация ECMAScript 5 определяет две функции, предназначенные для сериализации и восстановления объектов JavaScript: `JSON.stringify()` и `JSON.parse()`. В этих функциях используется формат обмена данными JSON (JavaScript Object Notation — запись объектов JavaScript, `http://json.org`). Синтаксис JSON напоминает синтаксис литералов объектов и массивов JavaScript.

```
o = {x:1, y:[false,null,""]};
```

```
s = JSON.stringify(o); // '{"x":1,"y":[false,null,""]}'
p = JSON.parse(s);     // Создание глубокой копии
```

Реализация этих функций в ECMAScript 5 довольно близка к реализации в ECMAScript 3, доступной по адресу `http://json.org/json2.js`. Для практических целей их можно считать одинаковыми. Можете свободно использовать функции ECMAScript 5 в ECMAScript 3 с модулем `json2.js`.

Обратите внимание на то, что синтаксис JSON является подмножеством синтаксиса JavaScript и поэтому не представляет всех значений JavaScript. Поддерживаются и могут быть восстановлены объекты, массивы, строки, конечные числа и значения `true`, `false` и `null`.

Методы чтения и записи свойств

Свойство содержит имя и значение. В ECMAScript 5 (и в последних реализациях ECMAScript 3 во всех основных браузерах, кроме IE) значение может быть заменено одним или двумя методами, которые называются *метод чтения свойства* (getter) и *метод записи свойства* (setter). Свойства, определенные методами чтения и записи (они называются *свойствами с методами доступа*), необходимо отличать от простых свойств, которые являются всего лишь значениями, напоминающими поля объектов в других объектно-ориентированных языках.

Когда программа читает значение свойства с методами доступа, интерпретатор запускает метод чтения (не передавая ему аргументы). Возвращаемое значение метода чтения становится значением выражения доступа к свойству. Когда программа записывает значение в свойство с методами доступа, интерпретатор вызывает метод записи и передает ему значение, возвращаемое с правой

стороны оператора присваивания. Возвращаемое значение метода записи игнорируется.

Определить свойство с методами доступа несложно с помощью объектного литерала.

```
var o = {
  // Обычное свойство (без методов доступа)
  data_prop: value,
  // Пара методов доступа
  get accessor_prop() {/*Возврат значения*/},
  set accessor_prop(value) {/*Запись значения*/}
};
```

Свойства с методами доступа определяются как одна или две функции, имена которых совпадают с именем свойства, но ключевое слово `function` заменено словом `get` или `set`. Обратите внимание на то, что для отделения имени свойства от метода доступа к свойству двоеточие не используется. После тела функции необходима запятая для отделения одного метода от другого. В качестве примера рассмотрим следующий объект, представляющий точку в двухмерном декартовом пространстве. Обычные свойства (без методов доступа) содержат координаты *x* и *y* точки, а свойства с методами доступа — эквивалентные полярные координаты точки.

```
var p = {
  x: 1.0,
  y: 1.0,
  // Не забудьте добавить запятые
  get r() {
    return Math.sqrt(this.x*this.x+this.y*this.y);
  },
  set r(newvalue) {
    var oldvalue = Math.sqrt(this.x*this.x+
                             this.y*this.y);
    var ratio = newvalue/oldvalue;
    this.x *= ratio;
```

```
    this.y *= ratio;
  },
  get theta() {return Math.atan2(this.y,this.x);}
};
```

Ниже приведен еще один пример полезного объекта, имеющего свойство с методами доступа.

```
// Генерация увеличивающихся серийных номеров
var serialnum = {
  // Это свойство содержит следующий номер
  // Символ $ обозначает закрытое свойство
  $n: 0,
  // Увеличение и возврат значения
  get next() { return this.$n++; },
  // Установка нового значения,
  // если оно больше текущего
  set next(n) {
    if (n >= this.$n) this.$n = n;
    else throw
     "серийный номер может только увеличиваться";
  }
};
```

Атрибуты свойств

Кроме имен и значений, у свойств есть атрибуты, которые определяют, являются ли свойства записываемыми, перечислимыми и конфигурируемыми. В ECMAScript 3 способа установки или изменения атрибутов не существует, все свойства являются записываемыми, перечислимыми и конфигурируемыми. Рассмотрим функции ECMAScript 5, позволяющие читать и устанавливать атрибуты свойств.

В данном разделе мы будем считать методы доступа к свойствам атрибутами свойств. Согласно этой логике можно даже сказать, что значение свойства без методов доступа является атрибутом, и мы так и будем полагать

в данном разделе. Следовательно, мы будем считать, что у свойства без методов доступа есть имя и четыре атрибута: *значение*, *доступность для записи*, *перечислимость* и *конфигурируемость*. Свойства с методами доступа не имеют атрибутов "значение" и "доступность для записи". Их доступность для записи определяется наличием или отсутствием метода записи свойства. Следовательно, у свойств с методами доступа есть четыре атрибута: *метод чтения*, *метод записи*, *перечислимость* и *конфигурируемость*.

В определенных в ECMAScript 5 методах чтения и установки атрибутов свойств используется объект, который называется *дескриптором свойства* и представляет набор из четырех атрибутов. Объект дескриптора свойства имеет четыре свойства с теми же именами, что и названия атрибутов представляемого свойства. Таким образом, объект дескриптора свойства без методов доступа имеет свойства `value` (значение), `writable` (доступный для записи), `enumerable` (перечислимый) и `configurable` (конфигурируемый). Дескриптор свойства с методами доступа имеет свойства `get` и `set` вместо `value` и `writable`. Свойства `writable`, `enumerable` и `configurable` являются булевыми значениями, а свойства `get` и `set` — функциями.

Чтобы получить дескриптор именованного свойства заданного объекта, нужно вызвать метод `Object.getOwnPropertyDescriptor()`.

```
// Возвращает {value: 1, writable:true,
//             enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor({x:1}, "x");
// Чтение свойства theta объекта p
// Возвращает {get: /*func*/, set:undefined,
//             enumerable:true, configurable:true}
Object.getOwnPropertyDescriptor(p, "theta");
```

Как видно из имени, метод `Object.getOwnPropertyDescriptor()` работает только с собственными свойствами. Для

чтения атрибутов унаследованных свойств нужно явно пройти по цепочке прототипов, как описано в следующем разделе.

Для установки атрибутов свойства или создания свойства с заданными атрибутами вызовите метод `Object.defineProperty()` и передайте ему изменяемый объект, имя изменяемого или создаваемого свойства и объект дескриптора свойства.

```
var o = {}; // Сначала свойств нет
// Добавление свойства
Object.defineProperty(o, "x", {value   1,
                               writable: true,
                               enumerable: false,
                               configurable: true});

// Проверка свойства
o.x; // => 1
Object.keys(o) // => []

// Изменение атрибута доступности для записи
Object.defineProperty(o, "x", { writable: false });

// Попытка изменить значение свойства
o.x = 2; // Ошибка TypeError в строгом режиме
o.x      // => 1

// Свойство все еще конфигурируемое, поэтому
// можно изменить его значение
Object.defineProperty(o, "x", { value: 2 });
o.x // => 2

// Сделаем свойство x свойством с методами доступа
Object.defineProperty(o, "x", {
      get: function() { return 0; }
});
o.x // => 0
```

Дескриптор свойства, передаваемый методу `Object.defineProperty()`, необязательно должен содержать все четыре атрибута. При создании свойства опущенные атрибуты принимают значения `false` или `undefined`. При модификации существующего свойства опущенные атрибуты просто остаются прежними. Не забывайте, что данный метод изменяет или создает только собственные свойства, а с унаследованными свойствами он не работает.

С помощью метода `Object.defineProperty()` можно создать или изменить более одного свойства за раз. Первый аргумент содержит модифицируемый объект, а второй аргумент — объект, отображающий имена создаваемых и модифицируемых свойств на дескрипторы этих свойств.

```
var p = Object.defineProperties({}, {
  x: { value: 1, writable: true,
       enumerable:true, configurable:true },
  y: { value: 1, writable: true,
       enumerable:true, configurable:true },
  r: {
    get: function() {
    return Math.sqrt(this.x*this.x+this.y*this.y)
     },
     enumerable:true,
     configurable:true
  }
});
```

В предыдущих разделах рассматривался метод `Object.create()`, определенный в ECMAScript 5. Первым аргументом этого метода должен быть объект прототипа создаваемого объекта. Данный метод принимает также второй аргумент, совпадающий с вторым аргументом метода `Object.defineProperties()`. Если передать набор дескрипторов свойств методу `Object.create()`, они будут применены для добавления свойств в создаваемый объект.

Атрибуты объекта

У каждого объекта есть атрибуты прототипа, класса и расширяемости.

prototype

Атрибут прототипа задает объект, от которого данный объект наследует свойства. Этот атрибут устанавливается при создании объекта. Как было указано в предыдущих разделах, объекты, созданные с помощью объектных литералов, получают в качестве прототипа объект `Object.prototype`. Объекты, созданные с помощью ключевого слова `new`, в качестве прототипа имеют свойство `prototype` функции конструктора. Объекты, созданные с помощью метода `Object.create()`, получают в качестве прототипа первый аргумент (он может быть равен `null`).

В реализациях ECMAScript 5 можно прочитать прототип любого объекта, передав данный объект методу `Object.getPrototypeOf()`. В ECMAScript 3 нет эквивалентного метода, но иногда можно получить прототип объекта `o` с помощью выражения `o.constructor.prototype`.

Чтобы выяснить, является ли один объект прототипом (или элементом цепочки прототипов) другого объекта, используйте метод `isPrototypeOf()`. Например, чтобы узнать, является ли `p` прототипом объекта `o`, запишите выражение `p.isPrototypeOf(o)`, как в следующем коде.

```
var p = {x:1};              // Определение прототипа
var o = Object.create(p); // Применение прототипа
p.isPrototypeOf(o)          // => true: o наследует от p
Object.prototype.isPrototypeOf(p) // => true для
                                  //любого объекта
```

Обратите внимание на то, что метод `isPrototypeOf()` играет роль, аналогичную оператору `instanceof`.

class

Атрибут класса содержит строку с информацией о типе объекта. Ни в ECMAScript 5, ни в ECMAScript 3 нельзя устанавливать этот атрибут. Кроме того, существуют только непрямые способы его чтения. Метод `toString()`, унаследованный от `Object.prototype`, возвращает строку в следующем формате:

```
[object класс]
```

Следовательно, для получения класса объекта можно вызвать метод `toString()` через этот объект, отсчитать восемь символов с начала и вернуть фрагмент строки. Главная сложность состоит в том, что многие объекты наследуют от других объектов более полезные варианты метода `toString()`, поэтому вызвать нужный вариант `toString()` можно только косвенно, с помощью метода `Function.call()`, как показано в главе 7. В листинге 5.2 определена функция, возвращающая класс переданного ей объекта.

Листинг 5.2. Функция, возвращающая класс объекта

```
function classof(o) {
  if (o === null) return "Null";
  if (o === undefined) return "Undefined";
  return Object.prototype.toString.call(o).slice(8,-1);
}
```

extensible

Атрибут `extensible` (расширяемый) задает, можно ли добавить в объект новое свойство. Спецификация ECMAScript 5 определяет функции чтения и установки атрибута расширяемости объекта. Чтобы выяснить, является ли объект расширяемым, передайте его методу

`Object.isExtensible()`. Чтобы сделать объект нерасширяемым, передайте его объекту `Object.preventExtensions()`.

Метод `Object.seal()`, как и `Object.preventExtensions()`, делает объект нерасширяемым, но, кроме того, делает все собственные свойства объекта неконфигурируемыми. Следовательно, после его вызова в объект нельзя добавить новые свойства, а существующие свойства нельзя удалить или сконфигурировать. Чтобы выяснить, вызывался ли метод `seal()`, необходимо вызвать метод `Object.isSealed()`.

Метод `Object.freeze()` блокирует объекты еще жестче. Он не только делает объект нерасширяемым, а его свойства неконфигурируемыми, но и переключает все собственные свойства, не имеющие методов доступа, в режим "только чтение". Это называется *замораживанием* объекта. Чтобы выяснить, заморожен ли объект, нужно вызвать метод `Object.isFrozen()`.

Важно помнить, что результаты вызова методов `Object.preventExtensions()`, `Object.seal()` и `Object.freeze()` необратимые. Кроме того, эти методы влияют только на передаваемый им объект, а на прототипы объекта они никак не влияют. И наконец, все эти три метода возвращают переданный им объект в новом состоянии, благодаря чему вызовы этих методов можно делать вложенными.

```
o = Object.seal(Object.create(Object.freeze(
      {x:1}), {y: { value: 2, writable: true}}));
```

Массивы

Массив — это упорядоченный набор значений. Каждое значение называется *элементом*, и каждый элемент имеет обозначенную числом позицию в наборе, которая называется *индексом*. Иными словами, индекс — это номер элемента в массиве. Массивы JavaScript *нетипизированные*. Элемент массива может иметь любой тип, а разные элементы одного массива могут иметь разные типы. Элемент массива может быть даже объектом или другим массивом. Это позволяет создавать сложные структуры данных, такие как массивы объектов и многомерные массивы. В JavaScript нумерация элементов массивов начинается с нуля, и применяются 32-битовые целочисленные индексы. Следовательно, первый элемент имеет номер 0, а максимальный номер элемента равен $2^{32}-2 = 4294967294$. Максимальное количество элементов составляет 4294967295. Массивы JavaScript *динамические*: при необходимости они могут уменьшаться или увеличиваться. Поэтому нет необходимости объявлять фиксированный размер массива при его создании, а при изменении размера нет необходимости повторно выделять память для массива. Каждый массив имеет свойство `length` (длина), возвращающее текущее количество элементов массива.

Массивы JavaScript — это объекты специального вида. Индексы напоминают целочисленные имена свойств, но на самом деле это нечто большее. Различные реализации интерпретаторов по-разному оптимизируют массивы, по-

этому обычно обращение к индексированным элементам массива выполняется намного быстрее, чем к свойствам объекта.

Объекты массивов наследуют свойства от объекта `Array.prototype`, который определяет мощный набор методов манипулирования массивами. Большинство этих методов *обобщенные* (generic). Это означает, что они правильно работают не только с истинными массивами, но и с "массивоподобными" объектами, такими как строки символов.

Создание массива

Легче всего создать массив с помощью литерала массива, который представляет собой заключенный в квадратные скобки список элементов, разделенных запятыми.

```
var empty = [];              // Пустой массив
var primes = [2, 3, 5, 7];   // Массив из четырех чисел
var misc = [{}, true, "a"]; // Элементы разных типов
```

Значения в литерале массива необязательно должны быть константами, но могут быть произвольными выражениями.

```
var base = 1024;
var table = [base, base+1, base+2, base+3];
```

Литерал массива может содержать объектные литералы и литералы других массивов.

```
var b = [[1,{x:1, y:2}], [2, {x:3, y:4}]];
```

Если в литерале массива две запятые находятся рядом без значения между ними, то элемент считается пропущенным, а массив называется *разреженным* (sparse). Пропущенный элемент имеет значение `undefined`.

```
var count = [1,,3];  // Элемент 1 не определен
count[1]             // => undefined
var undefs = [,,];   // Элементов нет, но длина = 2
```

Синтаксис литералов массивов разрешает добавить завершающую запятую. Это означает, что, например, массив `[1,2,]` состоит из двух, а не из трех элементов.

Еще один способ создания массива состоит в применении конструктора `Array()`, который можно вызвать одним из трех способов.

- Вызов без аргументов.

  ```
  var a = new Array();
  ```

- Создается пустой массив без элементов, эквивалентный литералу `[]`.

- Вызов с одним целочисленным аргументом, задающим длину (количество элементов) массива.

  ```
  var a = new Array(10);
  ```

  Будет создан массив заданной длины. Эту форму конструктора можно использовать для создания массива, когда заранее известно максимальное количество элементов. Обратите внимание на то, что в создаваемый массив при этом не записываются никакие значения, а индексные свойства массива `"0"`, `"1"` и последующие не определены.

- Явное задание нескольких элементов массива.

  ```
  var a = new Array(5, 4, 3, 2, 1, "testing");
  ```

  В этой форме аргументы конструктора становятся элементами нового массива.

Важно отметить, что применить литерал массива почти всегда проще, чем конструктор `Array()`.

Элементы и длина массива

Обращение к элементу массива выполняется с помощью оператора [ ]. Ссылка на массив должна находиться слева от квадратных скобок. В квадратных скобках необходимо поместить произвольное выражение, возвращающее (или преобразуемое в) целочисленное положительное значение. Этот синтаксис используется как для чтения, так и для записи значения элемента. Ниже приведены примеры правильных выражений.

```
var a = ["world"];     // Создание массива
var value = a[0];      // Чтение элемента 0
a[1] = 3.14;           // Запись элемента 1
i = 2;
a[i] = 3;              // Запись элемента 2
a[i + 1] = "Привет!"; // Запись элемента 3
a[a[i]] = a[0];        // Чтение и запись элементов
```

Не забывайте, что массивы — это объекты специального вида. Квадратные скобки, используемые для обращения к элементу массива, работают так же, как квадратные скобки, применяемые для обращения к свойству объекта. Интерпретатор преобразует заданные вами числовые индексы массива в строки. Например, индекс 1 становится строкой "1". После этого строка используется в качестве имени свойства.

У каждого массива есть свойство `length` (длина), причем наличие именно этого свойства отличает массивы от обычных объектов. Свойство `length` содержит количество элементов массива (предполагается, что опущенных элементов нет). Значение свойства `length` всегда на единицу больше, чем самый высокий индекс массива.

```
[].length             // => 0: массив не имеет элементов
['a','b','c'].length // => 3: наибольший индекс равен 2
```

Каждый массив является объектом, поэтому можно создать в нем свойство с произвольным именем. Фактически от обычных объектов массивы отличаются только тем, что при использовании имени свойства, являющегося целочисленным значением, меньшим $2^{32}-1$ (или преобразуемого в него), массив автоматически создает и заполняет свойство `length`.

Свойство `length` доступно для записи. Если присвоить ему положительное целое значение `n`, меньшее текущего значения, то все элементы массива с индексом, бо́льшим или равным `n`, будут удалены.

```
a=[1,2,3,4,5]; // Создание массива из 5 элементов
a.length = 3;  // Теперь массив такой: [1,2,3]
a.length = 0;  // Удаление всех элементов
a.length = 5;  // Опять 5 элементов, но пустых
```

Можно также присвоить свойству `length` значение, большее, чем текущая длина массива. В результате будут созданы пустые элементы, и массив станет разреженным.

Перечисление элементов массива

Пройти по элементам массива проще всего с помощью цикла `for` (см. главу 4).

```
var keys = Object.keys(o); // Массив имен свойств
var values = []            // Массив для значений свойств
for(var i = 0; i < keys.length; i++) {
var key = keys[i];         // Получение имен свойств
values[i] = o[key];        // Сохранение значений
}
```

Во вложенных циклах и в других ситуациях, в которых важна производительность, рекомендуется извлечь длину массива один раз перед началом цикла, чтобы не извлекать ее на каждой итерации.

```
for(var i = 0, len = keys.length; i < len; i++) {
  // Тело цикла
}
```

В ECMAScript 5 определен ряд новых методов, предназначенных для прохода по элементам массива путем передачи методу каждого элемента в порядке возрастания индексов функции, определяемой в коде. Наиболее общий из этих методов — `forEach()`.

```
var data = [1,2,3,4,5];  // Создание массива
var sumOfSquares = 0;    // Переменная-накопитель
data.forEach(function(x) {
   sumOfSquares += x*x;  // Накопление квадратов
         });
sumOfSquares             // =>55: 1+4+9+16+25
```

Многомерные массивы

Спецификация JavaScript не поддерживает истинные многомерные массивы, но их можно успешно имитировать, создавая массивы массивов. Для обращения к значению в массиве массива нужно всего лишь написать оператор `[]` два раза подряд. Предположим, что `matrix` — это массив массивов, содержащий числа. Для обращения к отдельному числу в этом массиве нужно написать `matrix[x][y]`. В приведенном ниже примере двухмерный массив используется для создания таблицы умножения, хорошо знакомой вам еще со школы.

```
// Создание многомерного массива
var table = new Array(10); // 10 строк таблицы
for(var i = 0; i < table.length; i++)
  table[i] = new Array(10); // 10 столбцов

// Инициализация массива
for(var row = 0; row < table.length; row++) {
```

```
  for(col = 0; col < table[row].length; col++) {
    table[row][col] = row*col;
  }
}

// Извлечение значения 5*7 из таблицы умножения
var product = table[5][7]; // => 35
```

Методы массивов

С объектами массивов ассоциированы многие полезные методы, рассматриваемые в данном разделе.

join()

Метод `Array.join()` преобразует все элементы массива в строки, выполняет их конкатенацию и возвращает полученную таким образом строку. Можно задать строку-разделитель, отделяющую элементы массива один от другого в результирующей строке. Если разделитель не задан, в этом качестве используется запятая.

```
var a = [1, 2, 3];
a.join(); // => "1,2,3"
a.join(" "); // => "1 2 3"
a.join(""); // => "123"
var b = new Array(5);
b.join('-') // => '----'
```

Метод `String.split()` выполняет обратную операцию: он создает массив, разбив полученную строку на части.

reverse()

Метод `Array.reverse()` изменяет последовательность элементов массива на обратную и возвращает "реверсированный" массив. Эта операция выполняется "на месте", т.е. метод не создает еще один массив с переупорядочен-

ными элементами, а переупорядочивает элементы в существующем массиве.

```
var a = [1,2,3];
a.reverse().join() // => "3,2,1"
a[0] // => 3: теперь [3,2,1]
```

sort()

Метод `Array.sort()` сортирует элементы массива "на месте" и возвращает отсортированный массив. Когда `sort()` вызван без аргументов, он сортирует элементы массива в алфавитном порядке.

```
var a = new Array("вишня", "яблоко", "апельсин");
a.sort();
var s = a.join(", ");
            // s == "апельсин, вишня, яблоко"
```

Если в массиве есть неопределенные элементы, то в процессе сортировки они перемещаются в конец массива.

Чтобы отсортировать массив в порядке, отличном от алфавитного, необходимо передать методу `sort()` функцию сравнения, которая определяет, какой элемент массива из двух полученных должен быть первым в отсортированной последовательности. Если первый аргумент должен находиться в отсортированной последовательности раньше второго, функция сравнения возвращает отрицательное число. Если же первый аргумент должен находиться после второго, она возвращает положительное число. Если два аргумента эквивалентны (т.е. их последовательность не играет роли), функция сравнения должна возвращать нуль. Например, чтобы отсортировать элементы массива не в алфавитном, а в числовом порядке, можно написать следующий код.

```
var a = [33, 4, 1111, 222];
```

```
a.sort();        // В алфавитном порядке: 1111,222,33,4
a.sort(function(a,b) {
                 // В числовом порядке: 4,33,222,1111
   return a-b;   // Возвращает <0, 0, or >0
  });
a.sort(function(a,b) {return b-a});
                      // Обратный числовой порядок
```

Выполнить сортировку в алфавитном порядке, не чувствительном к регистру, можно следующим образом.

```
a = ['жук', 'Лиса', 'кот']
a.sort();
 // Алфавитный порядок, чувствительный к регистру:
             ['Лиса','жук',кот']
a.sort(function(s,t) {
  var a = s.toLowerCase();
  var b = t.toLowerCase();
  if (a < b) return -1;
  if (a > b) return 1;
  return 0;
}); // => ['жук','кот','Лиса']
```

concat()

Метод `Array.concat()` **создает и возвращает новый массив, содержащий элементы исходного массива и значения аргументов, заданные при вызове. Если каждый из аргументов является массивом, то элементами результирующего массива становятся элементы массивов аргументов, а не массивы аргументов. Метод** `concat()` **не изменяет массив, через который он вызван. Ниже приведен ряд примеров.**

```
var a = [1,2,3];
a.concat(4, 5)          // Вернул [1,2,3,4,5]
a.concat([4,5]);        // Вернул [1,2,3,4,5]
a.concat([4,5],[6,7])  // Вернул [1,2,3,4,5,6,7]
a.concat(4, [5,[6,7]]) // Вернул [1,2,3,4,5,[6,7]]
```

slice()

Метод Array.slice() возвращает фрагмент указанного массива. Два аргумента задают начало и конец возвращаемого фрагмента. Результирующий массив содержит все элементы, начиная с заданного первым аргументом, вплоть до элемента, заданного вторым аргументом, но не включая его. Если передается только один аргумент, метод возвращает все элементы, начиная с указанной позиции до конца массива. Если один из аргументов отрицательный, он задает отсчет элементов массива, начиная с конца. Не забывайте, что метод slice() не изменяет массив, через который он вызван, а возвращает в качестве результата новый массив.

```
var a = [1,2,3,4,5];
a.slice  (0,3); // Возвращает [1,2,3]
a.slice(3); // Возвращает [4,5]
a.slice(1,-1); // Возвращает [2,3,4]
a.slice(-3,-2); // Возвращает [3]
```

splice()

Метод Array.splice() вставляет новый или удаляет существующий элемент массива. В отличие от slice() и concat(), этот метод изменяет массив, через который он вызван.

Первый аргумент splice() задает позицию в массиве, в которой нужно начать вставку или удаление элементов. Второй аргумент задает количество элементов, удаляемых из массива. Если второй аргумент опущен, удаляются все элементы, начиная с указанного первым аргументом, вплоть до конца массива. Метод возвращает массив удаленных элементов или пустой массив (если ни один элемент не удален).

```
var a = [1,2,3,4,5,6,7,8];
a.splice(4);    // Возвращает [5.6.7,8];
```

```
                // массив a теперь равен [1,2,3,4]
a.splice(1,2); // Возвращает [2,3];
                // массив a теперь равен [1,4]
a.splice(1,1); // Возвращает [4];
                // массив a теперь равен [1]
```

Первые два аргумента задают, какие элементы массива должны быть удалены. После них могут находиться дополнительные аргументы, которые задают элементы, вставляемые в массив, начиная с позиции, заданной первым аргументом.

```
var a = [1,2,3,4,5];
a.splice(2,0,'a','b');
  // =>[]; массив a равен [1,2,'a','b',3,4,5]
a.splice(2,2,3);
   // =>['a','b']; массив a равен [1,2,3,3,4,5]
```

Обратите внимание на то, что в отличие от `concat()` метод `splice()` вставляет массивы, а не их элементы.

push() и pop()

Методы `push()` и `pop()` позволяют работать с массивами таким образом, как будто это стеки. Метод `push()` присоединяет один или несколько элементов к концу стека (т.е. массива) и возвращает новую длину массива. Метод `pop()` выполняет обратную операцию: удаляет последний элемент стека, уменьшает длину массива и возвращает удаленное значение. Оба метода изменяют массив "на месте", не создавая измененную копию массива.

```
var stack = [];     // stack: []
stack.push(1,2);    // stack: [1,2], вернул 2
stack.pop();        // stack: [1], вернул 2
stack.push(3);      // stack: [1,3], вернул 2
stack.pop();        // stack: [1], вернул 3
stack.push([4,5]); // stack: [1,[4,5]], вернул 2
stack.pop()         // stack: [1], вернул [4,5]
stack.pop();        // stack: [], вернул 1
```

unshift() и shift()

Методы `unshift()` и `shift()` работают аналогично методам `push()` и `pop()`, но вставляют и удаляют элементы в начале, а не в конце массива. Метод `unshift()` вставляет элемент или элементы в начало массива и сдвигает существующие элементы в сторону более высоких индексов, чтобы освободить место для вставляемых элементов. Этот метод возвращает длину нового массива. Метод `shift()` удаляет первый элемент массива, сдвигает оставшиеся элементы влево и возвращает удаленный элемент. Ниже приведен ряд примеров.

```
var a = [];          // a:[]
a.unshift(1);        // a:[1], вернул 1
a.unshift(22);       // a:[22,1], вернул 2
a.shift();           // a:[1], вернул 22
a.unshift(3,[4,5]); // a:[3,[4,5],1], вернул 3
a.shift();           // a:[[4,5],1], вернул 3
a.shift();           // a:[1], вернул [4,5]
a.shift();           // a:[], вернул 1
```

toString()

Массив, как и любой объект JavaScript, имеет метод `toString()`. В случае массива этот метод преобразует каждый элемент в строку (вызвав метод `toString()` этого элемента) и возвращает список полученных строк, разделенных запятыми. Обратите внимание на то, что результирующая строка не содержит квадратных скобок или каких-либо иных разделителей значений элементов, кроме запятых.

```
[1,2,3].toString()          // => '1,2,3'
["a", "b", "c"].toString() // => 'a,b,c'
[1, [2,'c']].toString()     // => '1,2,c'
```

Методы массивов ECMAScript 5

В ECMAScript 5 определены девять новых методов массивов для прохода по элементам, преобразования, фильтрации, проверки, свертки и поиска. Большинство из этих методов принимают функцию в качестве первого аргумента и вызывают ее по одному разу для каждого (или по крайней мере для некоторых) элемента массива. В большинстве случаев предоставляемая вами функция вызывается с тремя аргументами: значением элемента массива, индексом элемента массива и самим массивом. Иногда необходим только первый аргумент, а второй и третий игнорируются. Большинство методов массивов, принимающих функцию через первый аргумент, принимают также необязательный второй аргумент. Если он задан, функция вызывается таким образом, будто она является методом второго аргумента. Таким образом, второй передаваемый вами аргумент становится в теле функции значением `this`. Возвращаемое значение функции может по-разному использоваться разными методами. Ни один из методов массивов, определенных в ECMAScript 5, не изменяет массив, через который он вызван, но функция, передаваемая методу массива может изменять массив.

forEach()

Метод `forEach()` проходит по массиву, вызывая заданную функцию для каждого элемента.

```
var data = [1,2,3,4,5]; // Сумма элементов
var sum = 0;            // Начинаем с нуля
data.forEach(function(value) { sum += value; });
sum // => 15

// Увеличение каждого элемента
data.forEach(function(v, i, a) { a[i] = v + 1; });
data // => [2,3,4,5,6]
```

map()

Метод map() передает заданной функции каждый элемент массива, через который он вызван, и возвращает новый массив, содержащий значения, возвращенные функцией. Следовательно, этот метод преобразует каждый элемент массива согласно заданному алгоритму.

```
a = [1, 2, 3];
b = a.map(function(x) { return x*x; });
// Теперь массив b равен [1, 4, 9]
```

filter()

Метод filter() возвращает массив, содержащий подмножество элементов исходного массива, через который он вызван. Передаваемая методу функция должна возвращать значение true или false. Если функция, получив элемент, возвращает true или значение, преобразуемое в true, то данный элемент включается в результирующее подмножество и добавляется в массив, который станет возвращаемым значением метода.

```
a = [5, 4, 3, 2, 1];
a.filter(function(x) { return x < 3 });
                                // => [2,1]
a.filter(function(x,i) { return i%2==0 });
                                // => [5,3,1]
```

every() и some()

Методы every() и some() являются предикатами массива: они применяют заданную функцию к элементам массива и возвращают значение true или false.

Метод every() напоминает квантор всеобщности ∀: он возвращает true, если заданная функция вернула true для каждого элемента массива.

```
a = [1,2,3,4,5];
// Все ли значения меньше 10?
a.every(function(x) { return x < 10; }) //=>true
// Все ли значения четные?
a.every(function(x) {return x%2 === 0;}) //=>false
```

Метод `some()` напоминает квантор существования ∃: он возвращает `true`, если существует хотя бы один элемент массива, для которого заданная функция возвращает `true`. Значение `false` будет возвращено, если заданная функция вернет `false` для всех элементов массива.

```
a = [1,2,3,4,5];
// Есть ли четное число?
a.some(function(x) { return x%2===0; }) //=> true
// Есть ли элемент, не являющийся числом?
a.some(isNaN) //=> false
```

Обратите внимание на то, что методы `every()` и `some()` прекращают проход по элементам массива, как только становится известным возвращаемое значение. При вызове через пустой массив метод `every()` возвращает значение `true`, а метод `some()` — значение `false`.

reduce() и reduceRight()

Методы `reduce()` и `reduceRight()` объединяют элементы массива с помощью заданной функции, возвращая единственное значение. В функциональном программировании это часто используемая операция, которую обычно называют *сверткой* (reduction).

```
var a = [1,2,3,4,5]
// Вычисление суммы элементов
a.reduce(function(x,y) { return x+y }, 0); // =>15
// Вычисление произведения элементов
a.reduce(function(x,y) { return x*y }, 1); // =>120
// Поиск наибольшего элемента
a.reduce(function(x,y) { return (x>y)?x:y; });
                                          // =>5
```

Метод `reduce()` принимает два аргумента. Первый — функция, выполняющая операцию свертки. Задача свертки состоит в том, чтобы каким-либо образом объединить, или свернуть, два значения в одно. Эта функция должна возвращать свернутое значение. В приведенных выше примерах функция выполняет свертку двух значений путем их суммирования, умножения или выбора большего значения. Второй (необязательный) аргумент — начальное значение, передаваемое функции.

Функции свертки, используемые в методе `reduce()`, существенно отличаются от функций, используемых в `forEach()` и `map()`. Значение элемента, индекс и массив сдвигаются и передаются через второй, третий и четвертый аргументы. В первом аргументе хранится аккумулированный результат процесса свертки. При первом вызове функции первый аргумент содержит начальное значение, передаваемое через второй аргумент методу `reduce()`. При следующих вызовах первый аргумент получает значения, получаемые при предыдущих вызовах. Например, при суммировании функция свертки в первый раз вызывается с аргументами `0` и `1`. Функция суммирует их и возвращает `1`. При втором вызове функция получает `1` и `2` и возвращает `3` и т.д. И наконец, окончательная сумма, равная `15`, возвращается методом `reduce()`.

Обратите внимание на то, что в третьем примере, посвященном поиску наибольшего элемента, передается только один аргумент, а начальное значение не задается. Если вызвать метод `reduce()` без начального значения, он применит в этом качестве первый элемент массива. Следовательно, при первом вызове первым и вторым аргументами будут считаться первый и второй элементы массива. Заметьте, что в примерах с суммированием и перемножением также можно было опустить начальные значения.

Метод `reduceRight()` работает так же, как `reduce()`, за исключением того, что обработка массива в нем выполняется наоборот: начинается с верхних индексов и продвигается к нижним.

`indexOf()` и `lastIndexOf()`

Методы `indexOf()` и `lastIndexOf()` ищут в массиве элемент с заданным значением и возвращают индекс его первого вхождения или `-1`, если он не найден. Метод `indexOf()` проходит массив от начала до конца, а метод `lastIndexOf()` — от конца к началу.

```
a = [0,1,2,1,0];
a.indexOf(1) // => 1: a[1] равен 1
a.lastIndexOf(1) // => 3: a[3] равен 1
a.indexOf(3) // => -1: такого элемента нет
```

В отличие от других методов, описанных в данном разделе, методы `indexOf()` и `lastIndexOf()` не принимают функцию в качестве аргумента. В них первый аргумент — искомое значение, а второй не обязателен: он задает индекс, начиная с которого следует выполнить поиск. Если он опущен, метод `indexOf()` начинает поиск с начала, а `lastIndexOf()` — с конца. Второй аргумент может быть отрицательным; в этом случае поиск начинается с другого конца, т.е. в первом случае — с конца, а во втором — с начала.

Тип `Array`

Выше уже упоминалось о том, что массивы являются объектами специального вида. Когда в программе встречается неизвестный объект (будем надеяться, не летающий), иногда бывает полезно выяснить, массив ли это. В реализациях ECMAScript 5 это можно сделать с помощью функции `Array.isArray()`.

```
Array.isArray([]) // => true
Array.isArray({}) // => false
```

В любой другой версии JavaScript это можно сделать с помощью следующей функции.

```
var isArray = Array.isArray || function(o) {
  var ts = Object.prototype.toString;
  return typeof o === "object" &&
         ts.call(o) === "[object Array]";
};
```

"Массивоподобные" объекты

Как указывалось выше, массивы — это объекты, имеющие специальное свойство `length` (длина). "Массивоподобный" объект — это обычный объект JavaScript со свойством `length` и числовыми свойствами, играющими роль индексов. На практике массивоподобные объекты встречаются редко (за исключением строк, которые тоже считаются массивоподобными объектами). Вызывать через них методы массивов нельзя. Кроме того, со свойством `length` не ассоциировано специальное поведение. В данном случае это обычное свойство. Тем не менее по элементам массивоподобного объекта можно проходить в цикле так же, как по элементам настоящего массива.

```
// Создание массивоподобного объекта
var a = {"0":"a", "1":"b", "2":"c", length:3};
// Проход по этому объекту как по массиву
var total = 0;
for(var i = 0; i < a.length; i++)
      total += a[i];
```

Многие алгоритмы массивов работают с массивоподобными объектами так же, как и с настоящими массивами, а многие методы массивов JavaScript специально

определены как обобщенные (generic) и могут правильно обрабатывать массивоподобные объекты. Эти объекты не наследуют прототип `Array.prototype`, поэтому вызывать методы массивов непосредственно через них нельзя. Однако их можно вызвать с помощью метода `call()` следующим образом (подробнее об этом – в главе 7).

```
// Создание массивоподобного объекта
var a = {"0":"a", "1":"b", "2":"c", length:3};
Array.prototype.join.call(a, "+") // => "a+b+c"
Array.prototype.map.call(a, function(x) {
       return x.toUpperCase();
})                          // => ["A","B","C"]
// Создание настоящего массива на основе
// массивоподобного объекта
Array.prototype.slice.call(a, 0) // => ["a","b","c"]
```

Некоторые браузеры определяют обобщенные функции массивов непосредственно через конструктор `Array`. В браузерах, поддерживающих эти функции, можно использовать код следующего вида.

```
var a = {"0":"a", "1":"b", "2":"c", length:3};
Array.join(a, "+")
Array.slice(a, 0)
Array.map(a,
     function(x) { return x.toUpperCase(); })
```

Строки в качестве массивов

В реализациях ECMAScript 5 (и многих браузерах предыдущих версий, включая IE8, выпущенных до появления ECMAScript 5) строки ведут себя так же, как массивы в режиме "только чтение". Это означает, что для обращения к отдельным символам строки можно использовать не только метод `charAt()`, но и квадратные скобки.

```
var s = test;
s.charAt(0) // => "t"
s[1]        // => "e"
```

Тем не менее оператор `typeof` возвращает значение `string`, а метод `Array.isArray()`, получивший имя строки, возвращает значение `false`. Поэтому отличить строку от массива несложно.

Главное преимущество индексации строк состоит всего лишь в том, что можно заменить вызовы метода `charAt()` выражениями с квадратными скобками, которые делают программу более компактной и легкой для визуального восприятия. Кроме того, тот факт, что строки ведут себя, как массивы, означает, что к ним можно применять обобщенные методы массивов, как в следующем примере. Приведенный ниже код удаляет из строки гласные буквы.

```
s = "Java"
Array.prototype.join.call(s,    ") // => "J a v a"
Array.prototype.filter.call(s, function(x) {
      return x.match(/[^aeiou]/); // Поиск согласных букв
   }).join("")          // => "Jv"
```

Функции

Функция — это блок кода JavaScript, который определен один раз, но может выполняться многократно. В разных языках программирования функции могут называться по-разному: методами, процедурами или подпрограммами. Функции JavaScript *параметризованные* — они могут содержать список параметров, значения которых определяются в вызывающей функции. В теле вызываемой функции параметры работают так же, как и локальные переменные. Выражение вызова функции предоставляет ей *аргументы* — значения, передаваемые в тело функции посредством параметров. Функция может возвращать значение, вычисленное в ее теле. В вызывающей функции возвращаемое значение присваивается выражению вызова. Кроме списка параметров, при вызове функция получает еще одно значение — *контекст вызова*. В теле функции контекстом вызова служит значение ключевого слова `this`.

Если функция включена в объект, она считается его свойством и называется *методом* этого объекта. Когда функция вызывается через некоторый объект, данный объект становится контекстом вызова, т.е. значением ключевого слова `this`. Функции, предназначенные для инициализации создаваемых объектов, называются *конструкторами*. Вы уже встречались с конструкторами в главе 5; более подробно этот вопрос рассматривается в главе 8.

В JavaScript функции являются объектами, поэтому ими можно манипулировать, как объектами в программе. Например, можно присвоить функцию переменной или передать ее другой функции. Как и для любого объекта, для функции можно создавать свойства и даже вызывать через нее методы.

Определение функции можно вложить в определение другой функции, причем количество уровней вложенности ничем не ограничено. Важно отметить, что тело вложенной функции входит в область видимости локальных переменных, определенных во внешней функции.

Определение функции

Функция определяется с помощью ключевого слова `function`, которое может находиться в выражении определения функции (см. главу 3) или в инструкции объявления функции (см. главу 4). В любой из этих форм определение функции начинается с ключевого слова `function`, после которого расположены следующие компоненты.

- Идентификатор, задающий имя функции. Имя является обязательной частью в объявлении функции. Оно используется как имя переменной, которой присваивается объект определяемой функции во время ее создания. В выражении определения функции имя не обязательно. Если оно есть, то оно ссылается на объект функции только в ее же теле.
- Список идентификаторов, заключенный в скобки. Идентификаторы разделены запятыми. Их количество может быть произвольным, включая нулевое. Включенные в список идентификаторы являются именами параметров функции. В теле функции они ведут себя так же, как и локальные переменные.

- Последовательность инструкций JavaScript, заключенная в фигурные скобки. Это тело функции, выполняемое при ее вызове.

В листинге 7.1 приведен ряд примеров определения функции в обеих формах: как выражения и как инструкции. Обратите внимание на то, что функция, определенная как выражение, полезна только как часть большего выражения (такого, как выражение присваивания или вызова), которое что-либо делает с определяемой функцией.

Листинг 7.1. Примеры определения функций JavaScript

```
// Вывод имени и значения каждого свойства
// объекта o; возвращаемое значение не определено
function printprops(o) {
   for(var p in o)
      console.log(p +      + o[p] + "\n");
}

// Вычисление расстояния между
// точками (x1,y1) и (x2,y2)
function distance(x1, y1, x2, y2) {
   var dx = x2 - x1;
   var dy = y2 - y1;
   return Math.sqrt(dx*dx + dy*dy);
}

// Рекурсивная функция (т.е. функция, которая
// вызывает саму себя), вычисляющая факториал.
// Напомним, что факториал x! равен произведению
// всех положительных целых чисел, которые меньше
// или равны x
function factorial(x) {
   if (x <= 1) return 1;
   return x * factorial(x-1);
}

// Это выражение определяет функцию, которая
```

```
// возводит в квадрат свой аргумент. Объект
// функции явно присвоен переменной square
var square = function(x) { return x*x; }

// Выражение функции может содержать имя, которое,
// однако, может быть полезным
// только для ее рекурсии
var f = function fact(x) {
   if (x <= 1) return 1;
   else return x*fact(x-1);
};

// Выражение функции можно использовать
// в качестве аргумента другой функции
data.sort(function(a,b) { return a-b; });

// Выражение функции, вызванное
// немедленно после определения
var tensquared = (function(x) {return x*x;}(10));
```

Обратите внимание на то, что при определении функции как выражения указывать имя функции необязательно. Выражение определения функции не объявляет переменную. В противоположность этому инструкция объявления функции фактически объявляет переменную и присваивает ей объект функции. Имя необходимо функции, чтобы на нее можно было ссылаться. В частности, рекурсивная функция вычисления факториала (см. пример выше) благодаря имени может ссылаться на саму себя. Если выражение определения функции содержит имя, то в локальную область видимости функции будет включено связывание этого имени с объектом функции. В результате имя функции становится локальной переменной, видимой в теле этой же функции. В большинстве случаев функция, определенная как выражение, не нуждается в имени. Его отсутствие делает запись более компактной. Выражение определения функции подходит для случаев,

когда функция используется только один раз, как в двух последних примерах, приведенных выше.

Как демонстрировалось в главе 4, инструкция объявления функции неявно "поднимается" в самую верхнюю часть сценария или внешней функции, поэтому функцию можно вызывать до того, как она объявлена. Однако это не верно для функции, определенной как выражение: чтобы вызвать такую функцию, нужно иметь возможность сослаться на нее, но нельзя сослаться на функцию, определенную как выражение, пока она не присвоена переменной. Поэтому функцию, определенную с помощью выражения, нельзя вызвать до ее определения.

Обратите внимание на то, что большинство (но не все) функций в листинге 7.1 содержат инструкцию `return` (см. главу 4). Эта инструкция приводит к прекращению выполнения тела функции, вычислению выражения, находящегося после ключевого слова `return`, и возврату в вызывающую функцию. Значение выражения после ключевого слова `return` становится значением выражения для вызова функции. Если после ключевого слова `return` нет ассоциированного с ним выражения, функция возвращает значение `undefined`. Если в теле функции нет инструкции `return`, она выполняется до конца (до последней закрывающей фигурной скобки) и возвращает в вызывающую функцию значение `undefined`.

Вложенные функции

В JavaScript функция может быть вложена в другую функцию, как в следующем примере.

```
function hypotenuse(a, b) {
  function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

В теле вложенной функции доступны параметры и локальные переменные всех внешних функций на всех уровнях вложенности. Например, в приведенном выше примере в теле функции `square()` можно читать и записывать параметры `a` и `b`, определенные в функции `hypotenuse()`. В следующих разделах этот вопрос рассматривается подробнее.

Как упоминалось в главе 4, инструкции объявления функций не являются "настоящими" инструкциями, потому что реализации ECMAScript 5 помещают их только на самом высоком уровне текущего контекста. Они могут находиться в глобальном коде или внутри других функций, но не внутри циклов, условных инструкций, блоков `try/catch/finally`, инструкций `with` или любых других блоков. Учтите, что это ограничение касается только функций, объявляемых как инструкции. Выражения определения функций могут находиться в любом месте кода JavaScript.

Выполнение функций

Код JavaScript, находящийся в теле функции, при ее определении не выполняется. Он выполняется при каждом ее вызове. Функцию можно вызвать одним из четырех способов:

- как функцию;
- как метод;
- как конструктор;
- через ее метод `call()` или `apply()`.

Вызов функции

Функция вызывается как функция или как метод с помощью выражения вызова (см. главу 3). Выражение вызо-

ва состоит из выражения функции, которое возвращает объект функции, и списка параметров, разделенных запятыми. Список параметров должен быть заключен в скобки. Количество параметров может быть равно нулю. Если выражение функции является выражением обращения к свойству (когда функция является свойством объекта или элементом массива), то оно считается выражением вызова метода (подробнее этот вопрос рассматривается в следующих разделах). Ниже приведено несколько примеров вызова обычных функций (не методов) с помощью выражений вызова.

```
printprops({x:1});
var total = distance(0,0,2,1) + distance(2,1,3,5);
var probability = factorial(5)/factorial(13);
```

При вызове вычисляется каждое выражение аргумента и результирующее значение присваивается соответствующему параметру функции. В теле функции ссылка на параметр возвращает значение аргумента.

При обычном вызове возвращаемое значение функции становится значением выражения вызова. Если функция завершается в результате того, что интерпретатор дошел до закрывающей фигурной скобки тела функции, она возвращает значение `undefined`. Если функция завершается вследствие выполнения оператора `return`, то возвращаемым значением служит значение выражения, записанного после ключевого слова `return`. Если непосредственно после слова `return` находится точка с запятой, то возвращается значение `undefined`.

Согласно ECMAScript 3 и в нестрогом режиме ECMAScript 5 контекстом вызова (значением `this`) служит глобальный объект. Однако в строгом режиме контекст вызова равен `undefined` (не забывайте, что это справедливо только для функций, но не для методов).

В функциях, вызываемых как функции, ключевое слово this обычно не используется. Однако его можно использовать для выяснения, выполняется ли функция в строгом режиме.

```
var strict = (function() { return !this; }());
```

Вызов метода

Метод — это функция JavaScript, определенная как свойство объекта. Если в коде определены функция f() и объект o, то можно определить метод m() объекта o с помощью следующей строки:

```
o.m = f;
```

Определив таким образом метод m() объекта o, можно вызвать его следующим образом:

```
o.m();
```

Если метод m() ожидает два аргумента, выражение вызова должно быть таким:

```
o.m(x, y);
```

Приведенное выше выражение вызова содержит выражение функции o.m и два выражения аргументов x и y. Выражение функции является выражением обращения к свойству (см. главу 3). Это означает, что функция вызывается как метод, а не как функция.

Аргументы и возвращаемое значение вызова метода обрабатываются точно так же, как и в случае стандартного вызова функции. Вызов метода отличается от вызова функции только одной важной особенностью: контекстом вызова. Выражение обращения к свойству состоит из двух частей: объекта (в данном примере — o) и метода (m). При использовании выражения вызова метода объ-

ект становится контекстом вызова, и код тела функции может ссылаться на этот объект с помощью ключевого слова this. Рассмотрим конкретный пример.

```
var calculator = { // Это объектный литерал
  operand1: 1,
  operand2: 1,
  add: function() {
    // Слово this ссылается на объект calculator
    this.result = this.operand1 + this.operand2;
  }
};
calculator.add(); // Вызов метода
calculator.result // => 2
```

В большинстве случаев для обращения (или доступа, что то же самое) к свойству используется точка, но в выражении доступа можно использовать и квадратные скобки, которые также приведут к вызову метода. Ниже приведены примеры обоих способов вызова метода.

```
o["m"](x,y); // Эквивалентно o.m(x,y).
a = [function(x) { return x+1 }];
a[0](z)      // Это также вызов метода
```

Для вызова метода можно использовать и более сложные выражения доступа к свойству.

```
// Вызов метода toUpperCase()
// через объект customer.surname
customer.surname.toUpperCase();
// Вызов метода m()
// через возвращаемое значение функции f()
f().m();
```

Обратите внимание на то, что this является только ключевым словом, но не именем переменной или свойства. Синтаксис JavaScript не позволяет присваивать this какое-либо значение.

В отличие от переменных, у ключевого слова this нет области видимости, поэтому вложенная функция не наследует значение this внешней функции. Если вложенная функция вызывается как метод, ее значением this является объект, через который она вызвана. Если вложенная функция вызывается как функция, то ее значением this служит либо глобальный объект (в нестрогом режиме), либо значение undefined (в строгом режиме). Многие считают (и это довольно распространенная ошибка), что при стандартном вызове функции ключевое слово this можно использовать для получения контекста вызова внешней функции. На самом же деле для получения доступа к значению this внешней функции необходимо сохранить его в переменной, доступной во вложенной функции. Традиционно для этого используется переменная с именем self, как в следующем примере.

```
var o = {             // Объект o
  m: function() {     // Метод m объекта o
    var self = this; // Сохранение значения this
    console.log(this === o); // Выводит "true"
    f(); // Вызов вложенной функции
    function f() {
    console.log(this === o); // Выводит "false"
    console.log(self === o); // Выводит "true"
    }
  }
};
o.m(); //
```

Вызов конструктора

Если перед выражением вызова метода или функции находится ключевое слово new, значит, это вызов конструктора (см. главы 3 и 5; более подробно конструкторы рассматриваются в главе 8). Вызов конструктора отлича-

ется от вызова метода или функции способом обработки аргументов, контекста вызова и возвращаемого значения.

Если выражение вызова конструктора содержит список аргументов в скобках, то выражения аргументов вычисляются и передаются в тело конструктора так же, как и в случае вызова метода или функции. Однако если список параметров пустой, то синтаксис JavaScript позволяет при вызове опустить также скобки. Например, следующие две строки эквивалентны.

```
var o = new Object();
var o = new Object;
```

Инструкция вызова конструктора создает пустой объект, который наследует свойство `prototype` конструктора. Обычно функция конструктора предназначена для инициализации создаваемого объекта. Вызываемый объект служит контекстом вызова, на который в теле конструктора можно ссылаться с помощью ключевого слова `this`. Обратите внимание на то, что новый объект служит контекстом вызова, даже если вызов конструктора выглядит так же, как и вызов метода. Это означает, что в выражении `new o.m()` объект `o` не является контекстом вызова.

Обычно функция конструктора не содержит ключевого слова `return`, потому что задача конструктора — инициализировать новый объект и завершить свою работу при достижении последней закрывающей фигурной скобки. В этом случае новый объект становится значением выражения для вызова конструктора. Однако если в конструкторе явно используется инструкция `return` для возвращения объекта, то этот объект становится значением выражения вызова. Если в конструкторе используется `return` без значения или возвращается примитивное значение, то возвращаемое значение игнорируется, и новый объект используется в качестве значения выражения вызова.

Косвенные вызовы

Функции JavaScript являются объектами и, как и любой объект, имеют методы. Два из этих методов — `call()` и `apply()` — позволяют вызвать функцию особым способом, косвенно. Первый аргумент, как у метода `call()`, так и у `apply()`, должен быть объектом, через который вызывается функция. Этот аргумент служит контекстом вызова и становится значением ключевого слова `this` в теле функции. Например, чтобы вызвать функцию `f()` как метод объекта `o` (не передавая аргументы), необходимо вызвать ее метод `call()` или `apply()`.

```
f.call(o);
f.apply(o);
```

Каждая из этих строк эквивалентна следующим трем строкам (предполагается, что у объекта `o` еще нет свойства `m`).

```
o.m = f;     // Временно делаем f методом объекта o
o.m();       // Вызываем m через o
delete o.m; // Удаляем m
```

В строгом режиме ECMAScript 5 первый аргумент метода `call()` или `apply()` становится значением `this`, даже если это значение `null` или `undefined` либо примитивное значение. В реализациях ECMAScript 3 и в нестрогом режиме значение `null` или `undefined` замещается глобальным объектом, а примитивное значение замещается оболочкой его объекта.

Все последующие аргументы `call()` после первого аргумента (контекста вызова) являются значениями, передаваемыми параметрам функции при вызове. Например, чтобы передать два числа в функцию `f()` и вызвать ее как метод объекта `o`, нужно написать такоке выражение:

```
f.call(o, 1, 2);
```

Метод `apply()` аналогичен методу `call()` за исключением того, что аргументы, передаваемые функции, задаются в массиве.

```
f.apply(o, [1,2]);
```

Если функция принимает неопределенное количество аргументов, метод `apply()` позволяет применить при вызове массив произвольной длины. Например, чтобы найти наибольшее значение в массиве чисел, можно использовать метод `apply()` для передачи элементов массива функции `Math.max()`.

```
var biggest = Math.max.apply(Math, array_of_numbers);
```

Обратите внимание на то, что метод `apply()` принимает как "массивоподобные" объекты, так и настоящие массивы. В частности, можно вызвать функцию с теми же аргументами, которые определены в текущей функции, передав массив `arguments` непосредственно методу `apply()`. Подробнее этот вопрос рассматривается в следующем разделе. Приведенный ниже код демонстрирует данный подход.

```
// Замена метода m объекта o версией, которая
// протоколирует сообщения до и после вызова
// исходного метода
function trace(o, m) {
  var original = o[m]; // Запоминаем метод
  o[m] = function() {  // Определяем новый метод
    console.log(new Date(), "Entering:", m);
    // Вызов исходного метода
    var result = original.apply(this, arguments);
    console.log(new Date(), "Exiting:", m);
    // Возврат результата исходного метода
    return result;
  };
}
```

Приведенная выше функция `trace()` получает имена объекта и метода и замещает указанный метод новым. Новый метод служит оболочкой для функциональности, дополняющей исходный метод.

Аргументы и параметры функции

Определение функции JavaScript не задает ожидаемые типы параметров функции. Соответственно, при вызове функции проверка типов передаваемых аргументов не выполняется. Фактически при вызове функции не проверяется даже количество передаваемых аргументов. В следующем разделе мы рассмотрим, что произойдет, если функция вызвана с меньшим или бо́льшим количеством аргументов, чем объявлено в ее определении.

Необязательные параметры

Когда функция вызвана с меньшим количеством аргументов, чем определено в списке параметров, "лишние" параметры получают значение `undefined`. На практике иногда полезно создать функцию, некоторые аргументы которой необязательные и могут быть опущены при вызове. Для этого нужно иметь возможность присвоить необязательным параметрам значения по умолчанию, как в следующем примере.

```
// Добавление имен перечислимых свойств объекта o
// в массив a и возврат массива "a"; если имя "a"
// опущено, функция создает и возвращает
// новый массив
function names(o, /* необязательный */ a) {
   if (a === undefined) // Если a опущено
      a = [];           // Новый массив
   for(var property in o) a.push(property);
   return a;
}
```

```
// Эту функцию можно вызвать
// с одним или двумя аргументами
var a = names(o); // Получение имен свойств
names(p,a); // Добавление свойств в массив
```

Как показано в главе 3, вместо инструкции `if` в первой строке функции `names()` можно использовать оператор `||` следующим образом.

```
a = a || [];
```

Список аргументов переменной длины: объект Arguments

Когда функция вызывается с бо́льшим количеством аргументов, чем определено имен параметров, то способа прямо сослаться на неименованные значения не существует. Решение этой проблемы предоставляет объект `Arguments`. В теле функции на объект `Arguments` данного вызова ссылается идентификатор `arguments`. Объект `Arguments` является массивоподобным объектом (см. главу 6), который позволяет передать функции значения аргументов, извлекаемые не по имени, а по номеру.

Предположим, определена функция `f`, которая ожидает один аргумент `x`. Если вызвать эту функцию с двумя аргументами, первый аргумент будет доступен в теле функции под именем параметра `x` или как элемент массива `arguments[0]`. Второй аргумент будет доступен только как элемент `arguments[1]`. Объект `Arguments` имеет свойство `length` (длина), определяющее количество его элементов. Следовательно, в теле функции `f`, вызванной с двумя аргументами, свойство `arguments.length` имеет значение 2.

Одно из главных преимуществ объекта `Arguments` — возможность создавать функции с произвольным количеством аргументов. Приведенная ниже функция принимает любое количество числовых аргументов и возвращает

значение наибольшего аргумента (аналогично встроенной функции `Math.max()`).

```
function max(/*     */) {
  var max = Number.NEGATIVE_INFINITY;
  // Поиск и запоминание наибольшего аргумента
  for(var i = 0; i < arguments.length; i++)
    if (arguments[i] > max) max = arguments[i];
  // Возврат наибольшего аргумента
  return max;
}
var largest = max(10, 100, 2, 4, 10000, 6);
                                    // => 10000
```

Обратите внимание на то, что функцию, принимающую произвольное количество аргументов, нельзя вызвать без аргументов. Кроме того, объект `arguments[]` можно использовать для создания функций, ожидающих фиксированное количество именованных обязательных аргументов, после которых следует произвольное количество неименованных необязательных аргументов.

Функции как пространства имен

Как упоминалось в главе 2, область видимости переменной, объявленной в теле функции, совпадает с телом функции (включая вложенные функции), но не распространяется за ее пределы. Переменные, объявленные вне функции, являются глобальными и видимы во всей программе. В JavaScript нет способа объявить переменную таким образом, чтобы она была видна только в одном блоке кода. По этой причине иногда полезно определить функцию только для того, чтобы создать временное пространство имен, в котором можно определить нужные переменные, не засоряя глобальное пространство имен.

Предположим, например, что у нас есть модуль кода JavaScript, который нужно использовать в нескольких программах (на стороне клиента, на веб-страницах, на сервере). Предположим также, что в этом коде определены переменные для хранения промежуточных результатов вычислений. Проблема состоит в том, что, поскольку данный модуль используется во многих программах (причем заранее неизвестно, в каких), мы не знаем, будут ли созданные им переменные конфликтовать с переменными, объявленными в других программах, в которых он используется. Решение данной проблемы состоит в помещении кода в функцию, которую затем можно вызывать в разных программах. Таким образом, переменные, которые иначе были бы глобальными, становятся локальными в функции.

```
function mymodule() {
   // Здесь поместите модуль кода.
   // Любые переменные данного модуля будут
   // локальными в данной функции,
   // что предотвратит их конфликты с глобальным
   // пространством имен
}
mymodule(); // Не забудьте вызвать функцию!
```

В этом коде определена только одна глобальная переменная: имя функции `mymodule`. Если же и этого слишком много, можете определить и вызвать анонимную функцию в единственном выражении, не добавив ни единой глобальной переменной.

```
(function() { // Именованное выражение
   // Здесь находится модуль кода
}()); // Завершение и вызов функции
```

Данная методика определения и вызова функции в единственном выражении применяется настолько ча-

сто, что стала канонической. Обратите внимание на использование скобок. Скобка перед ключевым словом `function` обязательна, потому что без нее интерпретатор примет ключевое слово `function` за инструкцию объявления функции. Когда же есть скобки, интерпретатор правильно считает, что это выражение определения функции. Для большей наглядности рекомендуется заключать в скобки функцию, вызываемую немедленно после определения, даже если скобки не нужны.

Замыкания

Как и в большинстве современных языков программирования, в JavaScript используются *лексические области видимости*. Это означает, что функции выполняются с использованием таких областей видимости, какими они были в момент определения, а не в момент вызова. Данная комбинация объекта функции с областью видимости, в которой она была определена, называется *замыканием* (closure). В коде JavaScript замыкания становятся нетривиальными при использовании вложенных функций. Существует ряд мощных методик программирования, в которых применяются замыкания вложенных функций. При первом знакомстве с замыканиями они кажутся довольно загадочными, поэтому важно хорошо понимать данную концепцию, чтобы уверенно ее применять.

Первый шаг к пониманию замыканий — знание правил лексических областей видимости вложенных функций. Рассмотрим следующий код.

```
var scope="Глобальная"; // Глобальная переменная
function checkscope() {
  var scope="Локальная"; // Локальная переменная
  function f() { return scope; }
  return f();
```

```
}
checkscope() // => "Локальная"
```

Функция `checkscope()` сначала объявляет локальную переменную, а затем определяет и вызывает функцию, которая возвращает значение этой переменной. Вам должно быть понятно, почему вызов `checkscope()` возвращает строку `Локальная`. Теперь немного изменим код. Можете ли вы сказать, что вернет функция `checkscope()()`?

```
var scope="Глобальная"; // Глобальная переменная
function checkscope() {
  var scope="Локальная"; // Локальная переменная
  function f() { return scope; }
  return f;
}
checkscope()() // Что она вернет?
```

В этом коде пара скобок "перекочевала" изнутри функции `checkscope()` наружу. Вместо вызова вложенной функции с возвратом ее результата, функция `checkscope()` теперь возвращает объект вложенной функции. Что произойдет, если вызвать эту вложенную функцию (с помощью еще одной пары скобок в последней строке кода) за пределами функции, в которой она определена?

Вспомним фундаментальное правило лексических областей видимости: при выполнении функций используется цепочка областей видимости, действительная в момент определения. Вложенная функция `f()` определена в цепочке, в которой переменная `scope` связана со строкой `Локальная`. Это связывание все еще остается в силе, когда функция `f()` выполняется, независимо от того, откуда она вызвана. Поэтому последняя строка кода вернет `Локальная`, а не `Глобальная`. В этом состоит удивительная и мощная природа замыканий: они захватывают привязки локальных переменных (или параметров) во внешней функции, в которой они определены.

Замыкания захватывают локальные переменные единственного вызова функции, что дает возможность использовать эти переменные в качестве закрытых. В приведенном ниже коде замыкание используется именно таким образом.

```
var uniqueInteger = (function() {
                    // Определение вызова
    var counter = 0;
          // Закрытая переменная функции
    return function() { return counter++; };
}());
```

Чтобы понять этот код, внимательно прочитайте его. Сначала вам покажется, будто первая строка кода присваивает функцию переменной `uniqueInteger`. Однако в действительности код определяет и вызывает (на что намекает открывающая скобка в первой строке) функцию, поэтому переменной `uniqueInteger` присваивается возвращаемое значение функции. Теперь, посмотрев на тело функции, мы увидим, что ее возвращаемым значением служит другая функция, а именно — объект вложенной функции, который присваивается переменной `uniqueInteger`. Вложенная функция имеет доступ к переменным в области видимости и может использовать переменную `counter`, определенную во внешней функции. После того как внешняя функция завершилась, никакой другой код не видит переменную `counter`. Доступ к ней имеет исключительно вложенная функция. Поэтому каждый вызов `uniqueInteger()` вернет новое целое число, и внешний код JavaScript не может изменить внутреннее значение `counter`.

Закрытые переменные наподобие `counter` необязательно доступны только одной функции. Можно определить две или больше вложенных функций в одной внешней функции, причем все они будут иметь доступ к одной и той же закрытой переменой. Рассмотрим следующий код.

```
function counter() {
  var n = 0;
  return {
    count: function() { return n++; },
    reset: function() { n = 0; }
  };
}
var c = counter(), // Создание двух счетчиков
    d = counter();
c.count() // => 0
d.count() // => 0: d и c подсчитываются независимо
c.reset() // у reset() и count() общее состояние
c.count() // => 0: поскольку c переустановлена
d.count() // => 1: d не была переустановлена
```

Функция `counter()` возвращает объект счетчика. Этот объект имеет два метода: `cont()` возвращает следующее целое число, а `reset()` переустанавливает внутреннее состояние. Важно понимать, что оба метода имеют доступ к закрытой переменной `n`. Кроме того, каждый вызов `counter()` создает новую цепочку областей видимости и новую закрытую переменную. Поэтому, если вызвать функцию `counter()` дважды, она вернет два объекта счетчика с разными закрытыми переменными. Вызов `count()` или `reset()` через один объект счетчика не влияет на другой объект счетчика.

В приведенном выше примере две функции определены в одной и той же цепочке областей видимости, поэтому они имеют доступ к одним и тем же закрытым переменным. Это весьма важная методика, но важно также понимать, когда замыкания могут ошибочно получить общий доступ к переменной, которая не должна быть для них общей. Рассмотрим следующий код.

```
// Эта функция возвращает функцию,
// которая всегда возвращает v
function constant(v) { return function()
                       { return v; }; }
```

```
// Создание массива функций constant()
var funcs = [];
for(var i = 0; i < 10; i++) funcs[i] = constant(i);

// Функция, находящаяся в пятом
// элементе массива, возвращает 5
funcs[5]() // => 5
```

Данный код создает в цикле много замыканий. При работе с таким кодом распространенная ошибка — попытка переместить цикл в функцию, которая определяет замыкания. Рассмотрим следующий код.

```
// Возвращает массив функций,
// которые возвращают 0-9
function constfuncs() {
  var funcs = [];
  for(var i = 0; i < 10; i++)
    funcs[i] = function() { return i; };
  return funcs;
}

var funcs = constfuncs();
funcs[5]() // Что она возвращает?
```

Приведенный выше код создает 10 замыканий и сохраняет их в массиве. Все замыкания определены в одном вызове функции, поэтому они имеют общий доступ к переменной `i`. Когда функция `consfuncs()` возвращается, значение переменной `i` равно 10, и все 10 замыканий видят данное значение. Следовательно, все функции в возвращенном массиве функций возвращают то же самое значение. Очевидно, это не то, что нам нужно. Важно помнить, что цепочка областей видимости, ассоциированная с замыканием, продолжает существовать. Вложенные функции не создают закрытых копий области видимости или статических снимков связывания переменной.

Кроме того, при создании замыканий важно помнить, что `this` — это ключевое слово, а не переменная. Как уже

упоминалось, с каждым вызовом функции ассоциировано значение this, и замыкание не имеет доступа к значению this внешней функции, если только внешняя функция не сохранила это значение в переменной.

```
var self = this; // Для использования
                 // во вложенной функции
```

Аналогично выполняется связывание переменной arguments. Это не ключевое слово языка. Переменная автоматически объявляется для каждого вызова функции. Замыкание создает собственное связывание для переменной arguments, поэтому оно не имеет доступа к массиву аргументов внешней функции, если только внешняя функция не сохранила массив в переменной с другим именем.

```
var outerArguments = arguments;
                 // Для вложенных функций
```

Свойства, методы и конструктор функции

Вы уже знаете, что функция — это объект в JavaScript-программе. Получив в качестве операнда функцию, оператор typeof возвращает строку function. Но функции — это не просто объекты, а объекты специального вида. Как и у любых других объектов, у них есть свойства и методы. Есть даже конструктор Function(), предназначенный для создания объектов функций. В предыдущих разделах рассматривались методы call() и apply() объекта функции, а в следующих разделах мы рассмотрим свойства, методы и конструктор Function(), принадлежащие функции.

Свойство length

В теле функции значение arguments.length определяет количество аргументов, переданных функции. Однако не путайте его со свойством функции length, которое имеет

совершенно другое назначение: это свойство, доступное только для чтения и определяющее *арность* (arity) функции, т.е. количество параметров, объявленных в списке. Чаще всего оно совпадает с количеством аргументов, ожидаемых функцией.

Свойство `prototype`

У каждой функции есть свойство `prototype`, которое ссылается на ее прототип. У разных функций объекты прототипа разные. Когда функция используется как конструктор, создаваемый объект наследует свойства объекта прототипа. Прототипы и свойство `prototype` рассматривались в главе 5. Более подробно мы вернемся к ним в главе 8.

Метод `bind()`

Метод `bind()` был добавлен в ECMAScript 5, но его легко имитировать в ECMAScript 3. Как видно из названия, главная цель метода — связывание функции с объектом. При вызове метода `bind()` через функцию `f` и передаче объекта `o` метод возвращает новую функцию. Вызов новой функции в качестве функции (а не метода) приводит к вызову исходной функции `f` как метода объекта `o`. Все аргументы, передаваемые новой функции, передаются исходной функции. Рассмотрим следующий пример.

```
// Нужно связать эту функцию
function f(y) { return this.x + y; }
var o = { x   1 }; // Связываемый объект
var g = f.bind(o); // После этого вызов g(x)
                   // приведет к вызову o.f(x)
g(2) // => 3
```

Реализовать связывание такого вида несложно с помощью следующего кода.

```
// Возврат функции, которая вызывает f
// как метод o, передавая ей все свои аргументы
function bind(f, o) {
  // Использование связанного метода
  if (f.bind) return f.bind(o);
  else return function() {
    // Если метод не связан, то связываем его
    return f.apply(o, arguments);
  };
}
```

Определенный в ECMAScript 5 метод `bind()` не только связывает функцию с объектом, но и решает еще одну задачу: связывает любой аргумент, переданный методу `bind()` после первого аргумента. Это часто используется в реальных программах. Следующий пример иллюстрирует данную концепцию.

```
var sum = function(x,y) { return x + y };
// Создание функции sum со значением this,
// связанным с null и первым аргументом,
// связанным со значением 1;
// эта функция ожидает только один аргумент
var succ = sum.bind(null, 1);
succ(2) // => 3:
        // x связана с 1, а y получает 2
```

Метод `toString()`

Как и любой объект JavaScript, каждая функция имеет метод `toString()`. Согласно спецификации ECMAScript 5 этот метод должен возвращать строку с инструкцией объявления функции. Однако на практике многие реализации метода `toString()` возвращают полный исходный код функции. Встроенные функции обычно возвращают строку, содержащую вместо тела функции что-то наподо-

бие "[native code]" (машинный код). Это означает, что исходный код метода недоступен.

Конструктор Function()

Обычно функцию определяют с помощью ключевого слова function либо в виде инструкции определения функции, либо как литеральное выражение функции. Но функцию можно определить и с помощью конструктора Function(), как в следующем примере.

```
var f = new Function("x", "y", "return x*y;");
```

Эта строка кода создает функцию, почти эквивалентную функции, определенной с помощью следующего, более знакомого вам синтаксиса.

```
var f = function(x, y) { return x*y; }
```

Конструктор Function() ожидает произвольное количество строковых аргументов. Последний аргумент — текст тела функции. Он может содержать любые инструкции JavaScript, разделенные точками с запятыми. Остальные аргументы конструктора являются строками, задающими имена параметров создаваемой функции. Если нужно определить функцию, не имеющую аргументов, передайте в конструктор одну строку, содержащую тело функции.

Важно отметить, что в функциях, создаваемых конструктором Function(), не используются лексические области видимости. Они всегда компилируются как функции верхнего уровня, в результате чего имеют доступ к глобальным, а не к локальным переменным.

Классы

Объекты JavaScript обсуждались в главе 5. Там каждый объект рассматривался как уникальный набор свойств, разный для каждого объекта. Однако часто полезно определить класс объектов, у которых есть ряд общих свойств. *Экземпляры* класса имеют разные значения свойств, которые описывают их состояние. Кроме того, у них есть ряд свойств, описывающих их поведение (эти свойства обычно являются *методами*). Поведение определено в классе и является общим для всех экземпляров класса. Рассмотрим, например, класс `Complex`, представляющий комплексные числа и методы, выполняющие арифметические операции над комплексными числами. Экземпляр класса `Complex` имеет два свойства, в которых хранятся действительная и мнимая части комплексного числа. Кроме того, в классе `Complex` должны быть определены методы, выполняющие операции сложения, вычитания, умножения и деления этих чисел.

В JavaScript классы основаны на механизме наследования прототипов. Если два объекта наследуют свойства одного и того же прототипа, то считается, что они являются экземплярами одного класса. Прототипы и наследование рассмотрены в главе 5. Если вы забыли, что такое прототипы, прочитайте эту главу еще раз, чтобы понимать материал данной главы.

Если два объекта наследуют один прототип, то обычно (но необязательно) они являются созданными и ини-

циализированными одной и той же функцией конструктора. Инициализаторы рассматривались в главе 3, а создание объектов и вызов конструкторов — в главе 7. Более подробно классы и конструкторы рассматриваются в следующих разделах.

Если вы знакомы со строго типизированными объектно-ориентированными языками, такими как Java или C++, вы наверняка заметите, что классы JavaScript существенно отличаются от классов в этих языках. Между ними есть некоторое синтаксическое сходство, поэтому несложно эмулировать многие средства "классических" классов средствами JavaScript. Тем не менее важно понимать, что механизм наследования классов и прототипов JavaScript существенно отличается от принципов наследования в Java и других аналогичных языках. Одно из важнейших отличий классов JavaScript состоит в том, что они динамически расширяемые, как будет показано далее.

Классы и прототипы

В JavaScript класс — это набор объектов, наследующих свойства одного и того же объекта-прототипа. Прототип — центральное средство класса (термины "прототип", "объект-прототип" и "объект прототипа" — синонимы). В листинге 5.1 была определена функция `inherit()`, которая возвращает новый объект, наследующий заданный прототип. В данной главе рассматривается встроенная функция ECMAScript 5 `Object.create()`, которую можно использовать вместо более универсальной функции `inherit()`. Определение класса JavaScript означает определение прототипа и применение функции `Object.create()` для создания объектов, наследующих данный прототип. Обычно новые экземпляры класса нужно инициализировать, поэтому довольно часто программисты пишут функцию, которая

создает и инициализирует новый объект. Данный процесс демонстрируется в листинге 8.1. В нем определены прототип объекта для класса, представляющего ряд значений, и функция, создающая и инициализирующая новый экземпляр класса.

Листинг 8.1. Простой класс JavaScript

```
// range.js: класс, представляющий диапазон

// Это создающая функция, возвращающая диапазон
function range(from, to) {
// Использование функции Object.create()
// для создания объекта, наследующего определенный
// ниже прототип. Прототип хранится как свойство
// данной функции, и в нем определены общие
// методы объктов класса
var r = Object.create(range.methods);
// Сохранение начальной и конечной точек
// диапазона. Они являются ненаследуемыми
// свойствами, уникальными для объекта
r.from = from;
r.to = to;
// Возвращение нового объекта
return r;
}

// Объект прототипа определяет методы, наследуемые
// всеми объектами range.
range.methods = {
// Возвращает true, если x попадает в диапазон
includes: function(x) {
return this.from <= x && x <= this.to;
},
// Вызов f() по одному разу для каждого числа.
// Метод работает только с числовыми диапазонами
foreach: function(f) {
for(var x=Math.ceil(this.from); x <= this.to; x++)
f(x);
},
```

```
// Возврат строки, представляющей диапазон
toString: function() {
  return "(" + this.from +       + this.to + ")";
}
};

// Пример использования объекта диапазона
var r = range(1,3);     // Создание диапазона
r.includes(2);          // => true: 2 попадает в диапазон
r.foreach(console.log); // Вывод 1 2 3
console.log(r);         // Вывод (1...3)
```

В листинге 8.1 определена функция `range()`, создающая объект диапазона. Свойство `range.methods` этой функции применяется как удобное место для хранения объекта, определяющего класс. Ни правила синтаксиса, ни соглашения не требуют этого. Кроме того, обратите внимание на то, что функция `range()` определяет свойства `from` (от) и `to` (до) для каждого объекта диапазона. Это не общие и не наследуемые свойства, они определяют уникальное состояние каждого объекта диапазона. И наконец, обратите внимание на то, что в наследуемых методах, определенных в `range.methods`, используются свойства `from` и `to`, причем для ссылки на них применяется ключевое слово `this`, ссылающееся на объект, через который вызван метод. Такое использование ключевого слова `this` — фундаментальная особенность методов любого класса.

Классы и конструкторы

В листинге 8.1 демонстрировался один из способов создания класса JavaScript. Однако данный способ не считается каноническим, потому что в нем нет определения конструктора. *Конструктор* — это функция, предназначенная для инициализации нового объекта. Конструкторы вызываются с помощью ключевого слова `new` (см. главу 7).

При вызове конструктора с помощью ключевого слова new автоматически создается объект, поэтому конструктору остается только инициализировать его состояние. Важная особенность вызовов конструктора состоит в том, что свойство prototype функции конструктора применяется в качестве прототипа нового объекта. Это означает, что все объекты, созданные определенным конструктором, наследуют один и тот же объект и, следовательно, являются членами одного класса. В листинге 8.2 показано, как изменить класс Range (см. листинг 8.1) с использованием функции конструктора вместо создающей функции.

Листинг 8.2. Создание объектов класса Range с помощью конструктора

```
// range2.js: класс, представляющий диапазон

// Это функция конструктора, который
// инициализирует новый объект Range.
// Обратите внимание на то, что он не создает и
// не возвращает объект, а только
// инициализирует его
function Range(from, to) {
// Сохранение значений начала и конца диапазона
// в ненаследуемых свойствах, уникальных для
// каждого объекта
this.from = from;
this.to = to;
}

// Все объекты Range наследуют данный объект.
// Имя свойства должно быть "prototype"
Range.prototype = {
// Возвращает true, если x попадает в диапазон
includes: function(x) {
return this.from <= x && x <= this.to;
},
// Вызов f() по одному разу
// для каждого целого в диапазоне
foreach: function(f) {
```

```
for(var x=Math.ceil(this.from); x <= this.to; x++)
f(x);
},
// Возврат строки, представляющей диапазон
toString: function() {
return "(" + this.from +       + this.to + ")";
}
};

// Пример использования объекта диапазона
var r = new Range(1,3); // Создание объекта range
r.includes(2);          // => true: 2 попадает в диапазон
r.foreach(console.log); // Вывод 1 2 3
console.log(r);         // Вывод (1...3)
```

Сравните листинги 8.2 и 8.1. Обратите внимание на различия между двумя способами определения класса. Преобразуя создающую функцию в конструктор, мы переименовали ее с `range()` на `Range()`. Это общепринятое соглашение: функция конструктора определяет класс, а имя класса принято начинать с буквы в верхнем регистре. Имена обычных методов и функций принято начинать с буквы в нижнем регистре.

Еще одно важное отличие: в конце данного примера функция конструктора `Range()` вызывается с помощью ключевого слова `new`, а создающая функция `range()` вызывается без него. В листинге 8.1 для создания объекта использовался обычный вызов функции (см. главу 7), а в листинге 8.2 для этого же применяется ключевое слово `new`. Поскольку конструктор `Range()` вызван с помощью ключевого слова `new`, для создания объекта не нужно вызывать метод `Object.create()` или выполнять какие-либо дополнительные операции. Новый объект автоматически создается перед вызовом конструктора и становится доступным как значение `this`. Конструктор `Range()` всего лишь инициализирует значение `this`. Конструктору не

нужно даже возвращать созданный объект. Выражение вызова конструктора автоматически создает объект, вызывает конструктор как метод этого объекта и возвращает новый объект.

Еще одно важное различие между листингами 8.1 и 8.2 — способ именования объекта прототипа. В первом примере прототип называется `range.methods`. Это интуитивно понятное и удобное имя, но оно произвольное. Во втором примере прототип называется `Range.prototype`, причем это обязательное имя. Выражение вызова конструктора автоматически задает использование объекта `Range.prototype` в качестве прототипа нового объекта класса `Range`.

И наконец, обратите внимание на общие черты листингов 8.1 и 8.2: методы диапазонов определены и вызываются одинаково в обоих случаях.

Идентичность классов и конструкторы

Как было показано выше, прототипы объектов являются основой идентичности класса, потому что два объекта являются экземплярами одного класса, только если они наследуют один и тот же объект прототипа. Функция конструктора, инициализирующая состояние нового объекта, не служит основой идентичности, потому что у двух разных функций конструкторов могут быть свойства `prototype`, указывающие на один прототип. Следовательно, оба этих конструктора можно использовать для создания экземпляров одного класса.

Несмотря на то что конструкторы не так фундаментальны, как прототипы, они все же служат "публичным лицом" классов. Имя функции конструктора обычно является именем класса. Например, конструктор `Range()` создает объекты `Range`. Важнее, однако, то, что конструк-

торы используются оператором instanceof для проверки членства объекта в классе. Например, чтобы узнать, является ли объект r членом класса Range, нужно выполнить следующий код.

```
// Возвращает true, если
// r наследует Range.prototype
r instanceof Range
```

Оператор instanceof фактически не проверяет, инициализировался ли объект r конструктором Range(). Он проверяет, наследует ли объект r прототип Range.prototype. Тем не менее синтаксис оператора instanceof побуждает использовать конструкторы для идентификации классов.

Свойство constructor

В листинге 8.2 свойству Range.prototype присвоен новый объект, содержащий методы класса. Определять методы как свойства одного объектного литерала довольно удобно, однако для создания объекта это необязательно. В качестве конструктора можно использовать любую функцию JavaScript, но для вызова конструктора необходимо свойство prototype. Следовательно, каждая функция JavaScript имеет свойство prototype. Значением этого свойства является объект, имеющий единственное неперечислимое свойство constructor. Значением свойства constructor является объект функции.

```
// F.prototype.constructor эквивалентно F
var F = function() {}; // Объект функции
var p = F.prototype;   // Объект прототипа
var c = p.constructor; // Функция прототипа
c === F                // => true
```

Существование предопределенного объекта прототипа со свойством constructor означает, что создаваемые объекты наследуют свойство constructor, ссылающее-

ся на их конструктор. Конструкторы идентифицируют класс, поэтому свойство `constructor` возвращает класс объекта.

```
var o = new F(); // Создание объекта o класса F
o.constructor === F // => true
```

На рис. 8.1 показаны отношения между функцией конструктора и объектом прототипа, а также обратная ссылка прототипа на конструктор и экземпляры класса, созданные с помощью конструктора.

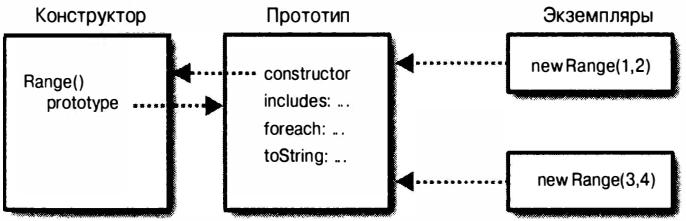


Рис. 8.1. Функция конструктора, ее прототип и экземпляры класса

Обратите внимание на то, что на рис. 8.1 конструктор `Range()` используется только для примера. Однако фактически класс `Range`, приведенный в листинге 8.2, переопределяет объект `Range.prototype` собственным объектом. Новый объект прототипа не имеет свойства `constructor`, поэтому экземпляры класса `Range`, определенного таким образом, также не имеют свойства `constructor`. Исправить эту проблему можно, явно добавив конструктор в прототип.

```
Range.prototype = {
constructor: Range, // Явная установка конструктора
includes: function(x) {
return this.from <= x && x <= this.to;
},
```

```
// и т.д.
};
```

Еще одна популярная методика состоит в использовании предопределенного объекта прототипа со встроенным свойством `constructor` и добавлении к этому объекту методов.

```
// Расширение предопределенного объекта
// Range.prototype, чтобы не переопределять
// свойство Range.prototype.constructor
Range.prototype.includes = function(x) {
  return this.from<=x && x<=this.to;
};
Range.prototype.foreach = function(f) {
  for(var x=Math.ceil(this.from); x <= this.to; x++)
    f(x);
};
Range.prototype.toString = function() {
  return "(" + this.from + "  " + this.to + ")";
};
```

Классы в стиле Java

Если вы программировали на Java или другом строго типизированном объектно-ориентированном языке, значит, вы знакомы с понятием *членов класса*, к которым относятся следующие сущности.

- **Поля экземпляра** — свойства или переменные, содержащие состояние индивидуального объекта.
- **Методы экземпляра** — методы, общие для всех экземпляров, но вызываемые через индивидуальные объекты.
- **Поля класса** — свойства или переменные, ассоциированные со всеми объектами данного класса, а не с одним объектом.

- **Методы класса** — методы, ассоциированные с классом, а не с экземпляром, и вызываемые через класс.

Одно из существенных отличий JavaScript от Java состоит в том, что в JavaScript функции являются значениями, и по этой причине жесткой границы между полями и методами не существует. Если значением свойства является функция, значит, свойство определяет метод; в противном случае это обычное свойство, т.е. "поле". Однако несмотря на это отличие, в JavaScript легко имитировать каждую из четырех категорий членов класса. В JavaScript для определения класса используются объекты трех разных видов (см. рис. 8.1), а свойства этих трех объектов играют роль следующих членов класса.

- **Объект конструктора.** Как указано выше, функция (объект) конструктора определяет имя класса JavaScript. Свойства, добавляемые программистом в объект конструктора, служат полями и методами класса.
- **Объект прототипа.** Свойства объекта прототипа наследуются всеми экземплярами класса. Свойства, значениями которых являются функции, ведут себя, как методы экземпляра класса.
- **Объект экземпляра.** Каждый экземпляр класса является полноценным объектом. Свойства, определенные непосредственно для экземпляра, не являются общими для всех экземпляров. Нефункциональные свойства, определенные для экземпляров, ведут себя, как поля экземпляров класса.

Алгоритм определения класса в JavaScript можно сократить до трех шагов. Первый — создание функции конструктора, которая устанавливает свойства экземпляра.

Второй — определение методов экземпляра в объекте `prototype` конструктора. И третий — определение методов и полей класса в конструкторе. Можно даже реализовать данный алгоритм как простую функцию `defineClass()`.

```
// Функция для определения классов
function defineClass(constructor, // Инициализация
                     methods,     // Методы экземпляра
                     statics)     // Свойства класса
{
  if (methods) {
    // Копирование методов в прототип
    for(var m in methods)
      constructor.prototype[m] = methods[m];
  }
  if (statics) {
   // Копирование статических свойств в конструктор
   for(var s in statics)
     constructor[s] = statics[s];
  }
  return constructor;
}

// Это простой вариант класса Range
var SimpleRange =
  defineClass(
    function(f,t) { this.f = f; this.t = t; },
    {
      includes: function(x) {
        return this.f <= x && x <= this.t;
    },
    toString: function() {
      return this.f + " " + this.t;
    }
  },
{
    upto: function(t) {
      return new SimpleRange(0, t);
    }
  }
);
```

Неизменяемые классы

В главе 5 рассматривался определенный в ECMAScript 5 метод `Object.defineProperties()`, предназначенный для определения неперечислимых свойств, доступных в режиме чтения. В этой же главе показано, что дескрипторы свойств можно передавать также в метод `Object.create()`. В данном разделе мы рассмотрим использование средств ECMAScript 5 для определения классов с неизменяемыми экземплярами. В листинге 8.3 приведена неизменяемая версия упомянутого выше класса `Range` с неперечислимыми методами экземпляров (как и методы встроенных классов). Интересная особенность листинга 8.3 состоит в том, что функция конструктора при вызове без ключевого слова `new` работает как создающая функция.

Листинг 8.3. Неизменяемый класс, содержащий неперечислимые методы

```
// Эта функция работает как с ключевым словом new,
// так и без него; в первом случае она играет роль
// конструктора, а во втором -- создающей функции
function Range(from,to) {
  // Это дескрипторы свойств,
  // доступных только для чтения
  var props = {
    from: { value:from, enumerable:true },
    to: { value:to, enumerable:true }
  };
  if (this instanceof Range) // Вызов конструктора
    Object.defineProperties(this, props);
  else // Вызов создающей функции
    return Object.create(Range.prototype, props);
}
// Установка прототипа с неперечислимыми свойствами
Object.defineProperties(Range.prototype, {
  includes: {
```

```
    value: function(x) {
      return this.from <= x && x <= this.to;
    },
    writable: true, configurable: true
  },
  foreach: {
    value: function(f) {
    for(var x=Math.ceil(this.from); x<=this.to; x++)
      f(x);
    },
    writable: true, configurable: true
  },
  toString: {
    value: function() {
    return "(" + this.from +       + this.to + ")";
    },
    writable: true, configurable: true
  }
});
```

Подклассы

В объектно-ориентированном программировании класс `B` может *расширять* класс `A`, при этом класс `B` называется *подклассом*, а класс `A` — *надклассом*. Экземпляры класса `B` наследуют все методы экземпляров класса `A`. В классе `B` можно определить собственные методы экземпляра; некоторые из них могут *переопределять* методы с этими же именами, определенные в классе `A`.

Ключевое условие создания подкласса в JavaScript — правильная инициализация объекта прототипа. Если объект `O` является экземпляром класса `B`, а `B` является подклассом класса `A`, то `O` должен наследовать свойства `A`. Однако для этого необходимо, чтобы объект прототипа `B` наследовал объект прототипа `A`. Решить данную задачу можно с помощью метода `Object.create()`, как показано ниже (можно также применить функцию `inherit()`; см. листинг 5.1).

```
// Подкласс B наследует надкласс A...
B.prototype = Object.create(A.prototype);
// ...и переопределяет наследуемый конструктор
B.prototype.constructor = B;
```

Две приведенные выше строки критичны для создания подклассов. Без них объект прототипа был бы обычным объектом, наследующим `Object.prototype`. Это означает, что создаваемый класс будет всего лишь подклассом класса `Object`, как и любой другой класс. Две приведенные выше строки необходимо добавить в функцию `defineClass()`, чтобы преобразовать ее в функцию определения подкласса `defineSubclass()`.

В листинге 8.4 класс `DateRange` определен как подкласс класса `Range`. Объекты даты в JavaScript можно сравнивать с помощью операторов < и >, поэтому класс `DateRange` наследует методы `includes()` и `toString()`, но переопределяет метод `foreach()` для перечисления дней в диапазоне. Обратите внимание на то, как установлено свойство `DateRange.prototype`, а также на то, что конструктор подкласса вызывает конструктор надкласса с помощью метода `call()` для инициализации нового объекта

Листинг 8.4. Подкласс класса Range

```
// Подкласс класса Range; наследует методы
// includes() и toString(); переопределяет метод
// foreach() для работы с датами
function DateRange(from, to) {
  // Использование конструктора надкласса
  // для инициализации подкласса
  Range.call(this, from, to);
}

// Эти строки ключевые для подкласса; прототип
// подкласса должен наследовать прототип надкласса
DateRange.prototype = Object.create(Range.prototype);
```

```
DateRange.prototype.constructor = DateRange;

// Это "статическое" поле подкласса содержит
// количество миллисекунд в сутках
DateRange.DAY = 1000*60*60*24;
// Вызов f() по одному разу
// для каждого дня в диапазоне
DateRange.prototype.foreach = function(f) {
  var d = this.from;
  while(d < this.to) {
    f(d);
    d = new Date(d.getTime() + DateRange.DAY);
  }
}

var now = new Date();
var tomorrow = new Date(now.getTime() + DateRange.DAY);
var nextweek = new Date(now.getTime() + 7*DateRange.DAY);
var week = new DateRange(now, nextweek);

week.includes(tomorrow) // => true
week.foreach(function(d) { // Вывод дней недели
console.log(d.toLocaleDateString());
});
```

Расширение классов

Встроенный в JavaScript механизм наследования на основе прототипов является динамическим. Это означает, что объект наследует свойства своего прототипа, даже если свойства прототипа изменились после создания объекта. Следовательно, классы можно расширять на этапе выполнения, добавляя новые методы в объекты прототипов. Ниже приведен код, добавляющий метод в класс Range.

```
// Возвращение нового диапазона
// с инвертированными конечными точками
```

```
Range.prototype.negate = function() {
  return new Range(-this.to, -this.from);
};
```

Прототипы встроенных классов JavaScript также "открытые". Следовательно, новые методы можно добавлять в классы чисел, строк, массивов, функций и т.п. Ниже приведен ряд примеров добавления методов.

```
// Функцию f() можно вызывать много раз, передавая
// номер итерации. Например, вывести
// "Привет" три раза можно так:
// var n = 3;
// n.times(function(n)
//       { console.log(n +   Привет"); });
Number.prototype.times = function(f, context) {
  var n = Number(this);
  for(var i = 0; i < n; i++) f.call(context, i);
};

// Определение метода ECMAScript 5 String.trim(),
// если его не существует; этот метод удаляет
// пробелы в начале и в конце строки
String.prototype.trim =
  String.prototype.trim || function() {
    if (!this) return this;
    return this.replace(/^\s+|\s+$/g, "");
  };

// Возврат имени функции или пустой строки "";
// если есть свойство с именем, метод извлекает
// его, в противном случае функция преобразуется
// в строку, из которой извлекается имя
Function.prototype.getName = function() {
  return this.name ||
    this.toString().
       match(/function\s*([^(]*)\(/)[1];
};
```

Добавление метода в `Object.prototype` делает его доступным для всех объектов. Однако делать так не рекомендуется, потому что в ECMAScript 5 добавленный таким способом метод нельзя сделать неперечислимым. Следовательно, он будет появляться в каждом цикле `for/in`, что весьма нежелательно, поскольку будет утрачен смысл использования циклов данного вида.

Регулярные выражения

Регулярное выражение — это объект, описывающий шаблон символов. В JavaScript регулярные выражения представлены классом `RegExp`. Кроме того, в классах `String` и `RegExp` есть методы, в которых регулярные выражения используются для выполнения мощных операций поиска и замены текстов путем сравнивания шаблонов. В данной главе рассматриваются синтаксис регулярных выражений, используемый для описания текстовых шаблонов, а также методы классов `String` и `RegExp`, в которых используются регулярные выражения.

Описание шаблонов с помощью регулярных выражений

В JavaScript каждое регулярное выражение является объектом класса `RegExp`. Конечно, его можно создать с помощью конструктора, однако такой способ громоздкий и применяется редко. Чаще регулярные выражения создаются с помощью специального литерального синтаксиса. Литерал регулярного выражения задается как строка символов, обрамленная символами косой черты (сравните со строковым литералом, который представляет собой строку символов, заключенных в кавычки). Рассмотрим следующую инструкцию.

```
var pattern = /s$/;
```

Эта строка создает объект класса RegExp и присваивает его переменной pattern. Данное регулярное выражение соответствует любой строке, заканчивающейся буквой s. С помощью конструктора RegExp() это же регулярное выражение можно определить следующим образом.

```
var pattern = new RegExp("s$");
```

Спецификация шаблона регулярных выражений определяет ряд символов. Большинство этих символов, включая все буквы и цифры, описывают просто соответствие самим себе. Например, регулярное выражение /java/ соответствует любой строке, содержащей подстроку java. Другие символы в регулярных выражениях не соответствуют себе, а имеют специальное значение. Например, выражение /s$/ содержит два символа. Первый символ (s) означает совпадение с собой, а второй — это специальный метасимвол, соответствующий концу строки. Следовательно, данное регулярное выражение соответствует любой строке, заканчивающейся символом s.

Литеральные символы

Почти все буквы и цифры в регулярном выражении трактуются буквально, т.е. соответствуют самим себе (такие символы называются *литеральными*). Некоторые специальные символы обозначаются специальными *управляющими последовательностями*, показанными в табл. 9.1.

Таблица 9.1. Управляющие последовательности, используемые в регулярных выражениях

Управляющая последовательность	Специальный символ
\0	Символ NUL (\u0000)
\t	Табулостоп (\u0009)

Управляющая последовательность	Специальный символ
`\n`	Новая строка (\u000A)
`\v`	Вертикальная табуляция (\u000B)
`\f`	Подача формы (\u000C)
`\r`	Возврат каретки (\u000D)
`\x`*nn*	Символ набора Latin, заданный шестнадцатеричным числом *nn*; например, `\x0A` — это то же самое, что `\t`
`\u`*xxxx*	Символ Unicode, заданный шестнадцатеричным числом *xxxx*; например, `\u0009` — это то же самое, что `\t`
`\c`*X*	Управляющий символ `^`*X*; например, `\cJ` — это то же самое, что `\n`

Приведенные ниже знаки пунктуации в регулярных выражениях имеют специальное значение.

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

Мы обсудим их в следующих разделах. Некоторые из этих символов имеют специальное значение только в определенном контексте регулярного выражения, а в других контекстах интерпретируются буквально, т.е. соответствуют самим себе. Однако, как правило, при необходимости включить символ пунктуации в регулярное выражение без специального значения нужно ввести перед ним символ обратной косой черты (\). Обратная косая черта отменяет специальное значение. Другие символы пунктуации, такие как одиночная кавычка или @, не имеют специального значения и в любом контексте регулярного выражения соответствуют самим себе.

Классы символов

Отдельные литеральные символы можно объединить в *класс символов*, разместив их в квадратных скобках. Класс символов соответствует любому символу, приведенному в квадратных скобках. Например, регулярное выражение `/[abc]/` соответствует букве `a`, `b` или `c`. Можно также определить отрицание класса символов. Отрицаемый класс соответствует любому символу, отличному от символов, перечисленных в квадратных скобках. Отрицание задается путем ввода символа `^` после левой квадратной скобки. Например, регулярное выражение `/[^abc]/` соответствует любому символу, отличному от `a`, `b` и `c`. Дефис в классе символов обозначает диапазон. Например, выражение `/[a-z]/` соответствует любой латинской букве в нижнем регистре, а выражение `/[a-zA-Z0-9]/` — любой латинской букве или цифре. Классы символов поддерживают также систему Unicode. Например, регулярное выражение `/[\u0400-\u04FF]/` соответствует любому символу кириллицы.

В синтаксисе регулярных выражений определены сокращения, или ярлыки, ряда часто используемых классов символов (табл. 9.2).

Таблица 9.2. Сокращения часто используемых классов символов

Выражение	Чему соответствует
`[...]`	Любой символ, приведенный в квадратных скобках
`[^...]`	Любой символ, отличный от приведенных в квадратных скобках
`.` (точка)	Любой символ, кроме конца строки или возврата каретки
`\w`	Любая буква или цифра ASCII, включая символ подчеркивания; эквивалентно `[a-zA-Z0-9_]`
`\W`	Любой символ, не являющийся буквой или цифрой ASCII; эквивалентно `[^a-zA-Z0-9_]`

Выражение	Чему соответствует
`\s`	Любой пробельный символ Unicode (пробел, неразрывный пробел, табулостоп и т.д.)
`\S`	Любой символ, не являющийся пробельным символом Unicode (обратите внимание на то, что `\w` и `\S` — это не одно и то же)
`\d`	Любая цифра ASCII; эквивалентно `[0-9]`
`\D`	Любой символ, отличный от цифры ASCII; эквивалентно `[^0-9]`
`[\b]`	Знак возврата на одну позицию назад (клавише Backspace)

Обратите внимание на то, что в определении класса символов можно использовать управляющие последовательности. Например, выражение `/[\s\d]/` соответствует пробельному символу или цифре.

Повторение

После символа или класса символов можно ввести символы повторения, задающие, сколько раз необходимо повторить указанные символы. Синтаксис повторения обобщен в табл. 9.3.

Таблица 9.3. Повторение в регулярных выражениях

Символы	Значение
`{`*n*`,` *m*`}`	Повторить как минимум *n*, но не более *m* раз
`{`*n*`, }`	Повторить *n* или более раз
`{`*n*`}`	Повторить точно *n* раз
`?`	Повторить предыдущий элемент нуль или один раз; это означает, что предыдущий элемент необязателен; эквивалентно `{0, 1}`
`+`	Повторить один или более раз; эквивалентно `{1, }`
	Повторить произвольное количество раз, включая нулевое; эквивалентно `{0, }`

Ниже приведен ряд примеров регулярных выражений, в которых задано повторение элементов.

```
/\d{2,4}/     // От 2 до 4 цифр
/\w{3}\d?/    // Три буквы и
              // одна необязательная цифра
/\s+java\s+/ // Слово "java" с пробелами
              // до и после него
/[^(]*/       // Нуль или более символов,
              // не являющихся открывающей скобкой
```

Будьте осторожны с символами * и ?. Они могут означать повторение предыдущего элемента нуль раз, поэтому им разрешено не соответствовать ничему. Например, регулярное выражение /a*/, как ни странно, может соответствовать строке bbbb, потому что в ней нет символа a.

Нежадное повторение

Символы повторения, приведенные в табл. 9.3, задают повторение произвольное количество раз, при этом они "захватывают" все подходящие части обрабатываемой строки. Такое повторение называется *жадным*. Можно также определить *нежадное* повторение. Для этого достаточно поместить знак вопроса после символа или символов повторения: ??, +?, *? или даже {1,5}?. Например, регулярное выражение /a+/ соответствует одному или нескольким вхождениям символа a. Если применить его к строке aaa, оно охватит все три символа a. Выражение /a+?/ также охватывает произвольное количество символов a, но как можно меньшее. Если применить его к этой же строке (aaa), оно охватит только первый символ a.

Альтернативы, группировка и ссылки

В регулярных выражениях можно использовать специальные символы, задающие альтернативные варианты, группировку подвыражений и ссылки на предыдущие

подвыражения. Символ | разделяет альтернативные варианты. Например, выражение /ab|cd|ef/ соответствует строке ab, cd или ef. Еще пример: выражение /\d{3}|[a-z]{4}/ соответствует либо трем цифрам, либо четырем буквам в нижнем регистре.

Учитывайте, что альтернативы перебираются слева направо, пока не будет найдено совпадение. Если левая альтернатива соответствует обрабатываемой строке, правая альтернатива игнорируется, даже если соответствие "лучшее". Следовательно, выражение /a|ab/, примененное к строке ab, соответствует первому символу a.

Скобки имеют в регулярных выражениях ряд специальных значений. Одно из них – группировка отдельных элементов (т.е. создание подвыражения) таким образом, что они будут интерпретироваться символами |, *, + и ? как один элемент. Например, выражение /java(script)?/ соответствует слову java, после которого находится необязательное слово script. Выражение /(ab|cd)+|ef/ соответствует либо строке ef, либо повторению один или более раз строки ab или cd.

Еще одно назначение скобок – определение подшаблона в шаблоне. Когда регулярное выражение успешно находит соответствие в целевой строке, то можно извлечь части целевой строки, соответствующие любому заданному подшаблону, заключенному в скобки (получение подстрок рассматривается далее). Предположим, что нужно найти одну или несколько букв в нижнем регистре, после которых есть одна или несколько цифр. Для этого можно применить шаблон /[a-z]+\d+/. Но предположим, что нас интересуют только цифры в конце каждого соответствия. Заключим часть шаблона в скобки: /[a-z]+(\d+)/. После этого можно будет извлечь цифры из подшаблона, как показано далее.

Использование подвыражений в скобках позволяет ссылаться на подвыражение далее в том же регулярном выражении. Ссылка начинается символом \, после которого необходимо поместить цифру или цифры. Число указывает на номер подвыражения в скобках внутри регулярного выражения. Например, шаблон \3 ссылается на третье подвыражение.

Ссылка на предыдущее подвыражение регулярного выражения возвращает не шаблон подвыражения, а действительный текст, соответствующий данному шаблону. Следовательно, ссылки можно использовать для наложения ограничений, задающих части строки с одинаковыми символами. Например, следующее регулярное выражение соответствует нулевому или большему количеству символов, заключенных в одинарные или двойные кавычки.

```
/['"][^'"]*['"]/
```

Однако типы открывающей и закрывающей кавычек могут не совпадать. Например, может получиться, что открывающая кавычка одинарная, а закрывающая — двойная. Для решения этой проблемы необходимо использовать ссылку на предыдущее соответствие.

```
/(['"])[^'"]*\1/
```

Ссылка \1 соответствует тексту, найденному первым подвыражением в скобках. В данном примере выражение задает заключение строки в однотипные кавычки.

Можно также группировать элементы регулярного выражения, не создавая числовые ссылки на них. Для этого после открывающей скобки нужно ввести знак вопроса и двоеточие, т.е. группа выглядит не как (...), а как (?:...). Правила применения альтернативных вариантов, группировки и создания ссылок приведены в табл. 9.4.

Таблица 9.4. Альтернативы, группы и ссылки в регулярных выражениях

Символы	Назначение
|	Альтернативные варианты; выполняется поиск соответствия подвыражению слева или справа от данного символа
(. . .)	Группировка нескольких элементов в один элемент, к которому может быть применен символ *, +, ? или |. Символы, возвращенные группой, могут использоваться для ссылки далее в регулярном выражении
(?: . . .)	Только группировка; возвращенные символы не используются в ссылках и, следовательно, не имеют номера
\n	Ссылка, т.е. подстановка символов, возвращенных группой номер *n*. Группы являются подвыражениями (возможно, вложенными) и нумеруются слева направо. Если группы вложенные, нумеруются открывающие скобки. Группы, начинающиеся символами (?: в нумерации пропускаются

Задание позиции соответствия

Как было показано выше, многие элементы регулярных выражений находят соответствие одному символу в строке. Например, выражение `\s` соответствует одному пробельному символу. Однако некоторые элементы регулярных выражений соответствуют позициям между символами, а не символам. Например, `\b` соответствует границе слова, т.е. позиции между `\w` (буква ASCII) и `\W` (символ, отличный от буквы). Такие элементы, как `\b`, задают не символы, а позиции, в которых найдено соответствие. Иногда эти элементы называют *якорями* (anchors), потому что они "закрепляют" шаблон в определенной позиции целевой строки. В регулярных выражениях часто используются якорные элементы `^` (начало строки) и `$` (конец строки).

Например, чтобы задать соответствие строке, содержащей только слово `JavaScript`, необходимо написать

/^JavaScript$/. Если нужно найти отдельное слово Java (не входящее в слово JavaScript или какое-либо другое слово), примените шаблон /\sJava\s/, требующий пробелов в начале и конце слова. Однако данное решение порождает две проблемы. Во-первых, соответствия не произойдет, если Java находится в начале или конце строки или в конце предложения. Вокруг Java обязательно должны быть пробельные символы. Во-вторых, данный шаблон возвращает слово Java с пробелами, а это, видимо, не то, что обычно нужно. Поэтому вместо соответствия пробельным символам лучше применить шаблон соответствия границе слова \b. Тогда регулярное выражение будет выглядеть так: /\bJava\b/. Якорный элемент \B определяет позицию, не являющуюся границей слова. Таким образом, шаблон /\B[Ss]cript/ соответствует словам JavaScript и postscript, но не script и Scripting.

Виды якорных выражений перечислены в табл. 9.5.

Таблица 9.5. Элементы якорных выражений

Элемент	Назначение
^	Начало строки (в многострочном или однострочном тексте)
$	Конец строки (в многострочном или однострочном тексте)
\b	Граница слова, т.е. позиция между \w и \W или между \w и концом строки (однако [\b] соответствует символу возврата — клавише <Backspace>)
\B	Позиция, не являющаяся границей слова
(?=*p*)	Положительное условие на последующие символы. Требует, чтобы последующие символы соответствовали шаблону *p*, но не включает эти символы в найденную строку
<?!*p*)	Отрицательное условие на последующие символы. Требует, чтобы последующие символы не соответствовали шаблону *p*

Флажки

Флажки регулярных выражений задают высокоуровневые правила соответствия шаблонам. В отличие от других элементов регулярных выражений, флажки находятся не между символами косой черты, а после второй косой. В JavaScript поддерживаются три флажка. Флажок `i` задает поиск соответствий, нечувствительных к регистру букв. Флажок `g` задает глобальный поиск соответствий, т.е. найдены должны быть все соответствия в целевой строке. Флажок `m` задает поиск соответствий в многострочном режиме. Это означает, что если целевая строка содержит символы новой строки, то якоря `^` и `$` соответствуют началу и концу каждой строки и, кроме того, началу и концу всего текста. Флажки могут задаваться в любой комбинации. Например, шаблон `/java$/im` соответствует строкам `java` и `Java\nis fun`.

Все флажки регулярных выражений перечислены в табл. 9.6. Флажок `g` подробнее рассматривается в следующем разделе.

Таблица 9.6. Флажки регулярных выражений

Флажок	Назначение
`i`	Задание соответствий, нечувствительных к регистру букв
`g`	Глобальный поиск соответствий; это означает, что процедура поиска не останавливается, найдя первое соответствие, а ищет все соответствия
`m`	Многострочный режим; символ `^` соответствует началу каждой строки и всего текста, а символ `$` — концу каждой строки и всего текста

Использование регулярных выражений

В данном разделе рассматриваются методы объектов `String` и `RegExp`, в которых регулярные выражения исполь-

зуются для поиска соответствий шаблонам и выполнения операций поиска и замены.

Методы класса String

Класс String поддерживает четыре метода, использующих регулярные выражения. Самый простой метод — search(). Он вызывается через строку, принимает регулярное выражение и возвращает либо позицию первого символа в начале первого соответствия, либо -1, если соответствий нет. Например, приведенный ниже код возвращает 4.

```
"JavaScript".search(/script/i);
```

Метод search() не выполняет глобальный поиск; в аргументе регулярного выражения он игнорирует флажок g.

Метод replace() выполняет операции поиска и замены. Через первый аргумент он принимает регулярное выражение, а через второй — подставляемую строку. Метод ищет соответствия в строке, через которую он вызван. Если в регулярном выражении установлен флажок g, метод заменяет все найденные соответствия; в противном случае метод заменяет только первое соответствие. Если через первый аргумент передается не регулярное выражение, а строка, то метод replace(), в отличие от метода search(), не преобразует строку в регулярное выражение с помощью конструктора RegExp(), а ищет точное совпадение заданной строки. Например, в следующей инструкции метод replace() применяется для унификации регистров букв JavaScript во всем объекте text.

```
text.replace(/javascript/gi, "JavaScript");
```

Однако метод replace() еще более мощный. Как вы помните, подвыражения в скобках нумеруются слева направо, и регулярное выражение запоминает подстроки,

возвращаемые каждым подвыражением. Если в подставляемой строке после символа $ находится цифра, метод заменит эти два символа возвращенной подстрокой с указанным номером. Данное средство можно использовать, например, для замены прямых двойных кавычек, у которых открывающая и закрывающая кавычки не отличаются одна от другой, типографскими кавычками « и ».

```
var quote = /"([^"]*)"/g;
text.replace(quote, '«$1»');
```

Второй аргумент метода `replace()` может быть функцией, которая динамически вычисляет подставляемую строку. Если передать функцию, она будет вызвана по одному разу для каждого соответствия. Первый аргумент функции — строка, соответствующая регулярному выражению, а остальные аргументы — тексты, соответствующие каждому подвыражению шаблона. Возвращаемое выражение функции применяется в качестве подставляемой строки.

Наиболее общий метод `match()` принимает единственный аргумент — регулярное выражение и возвращает массив, содержащий результаты поиска соответствий. Если в регулярном выражении установлен флажок `g`, метод возвращает массив всех соответствий, обнаруженных в строке. Например, приведенная ниже инструкция находит все цифры и создает строковый массив, содержащий эти цифры в порядке вхождения.

```
"1 плюс 2 равно 3".match(/\d+/g)  // => ["1", "2", "3"]
```

Если в регулярном выражении нет флажка `g`, метод `match()` не выполняет глобальный поиск, а просто возвращает первое соответствие. Однако и в этом случае метод `match()` возвращает массив: его первый элемент содержит соответствующую строку, а остальные элементы — подстроки, соответствующие подвыражениям регулярного

выражения. По аналогии с методом `replace()` можно считать, что `a[n]` содержит `$n`.

Рассмотрим пример синтаксического разбора строки с URL-адресом с помощью следующего кода.

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Адрес http://www.example.com/~david";
var result = text.match(url);
if (result != null) {
  var fullurl = result[0];   // полное соответствие
  var protocol = result[1]; // => "http"
  var host = result[2];      // => "www.example.com"
  var path = result[3];      // => "~david"
}
```

И наконец, рассмотрим последний метод класса `String`, в котором используются регулярные выражения: метод `split()`. Он разбивает строку, через которую он вызван, на массив подстрок, используя аргумент в качестве разделителя.

```
"123,456,789".split(","); //=>["123","456","789"]
```

Метод `split()` может также принимать регулярное выражение через аргумент. Это позволяет, например, задать символ разделителя таким образом, чтобы были удалены смежные пробельные символы.

```
"1   2,3 ".split(/\s*,\s*/); // => ["1","2","3"]
```

Свойства и методы класса RegExp

Каждый объект `RegExp` имеет пять свойств. Свойство `source` содержит текст регулярного выражения. Свойство `global` задает, установлен ли флажок `g`. Свойство `ignoreCase` задает флажок `i`, а свойство `multiline` — флажок `m`. И наконец, целочисленное свойство `lastIndex`, доступное для чтения и записи, используется для шаблонов с флажком `g` и содержит позицию в строке, с которой начинается сле-

дующий шаг поиска. Свойство lastIndex используется методами exec() и test(), как описано далее.

В объекте RegExp определены два метода, выполняющие поиск соответствия шаблонам. Они работают аналогично описанным выше методам класса String. Метод exec() класса RegExp напоминает метод match() класса String (см. выше) за исключением того, что метод exec() принимает строку, а метод match() принимает регулярное выражение. Метод exec() выполняет регулярное выражение в заданной строке. Если он не находит соответствия, он возвращает значение null. Если соответствие найдено, он возвращает массив, аналогичный возвращаемому методом match() при отсутствии флажка g. Нулевой элемент массива содержит строку, соответствующую регулярному выражению, а каждый последующий элемент содержит подстроку, соответствующую очередному подвыражению. В свойстве index находится позиция первого символа, в которой найдено соответствие, а свойство input ссылается на исходную строку.

В отличие от match(), метод exec() возвращает один и тот же массив независимо от наличия флажка g в регулярном выражении. Как вы помните, метод match() возвращает массив совпадений при обработке глобального регулярного выражения. В противоположность этому метод exec() всегда возвращает одно соответствие и предоставляет полную информацию о нем. При вызове метода exec() для регулярного выражения с флажком g свойство lastIndex объекта регулярного выражения получает позицию символа, находящегося непосредственно после соответствующей подстроки. При вызове метода exec() второй раз для этого же регулярного выражения он начинает поиск с позиции, указанной в свойстве lastIndex. Если метод не находит соответствия, он устанавливает свойство

lastIndex равным нулю (программно можно изменить значение lastIndex в любой момент). Такое поведение позволяет вызывать метод exec() многократно для прохода в цикле по всем соответствиям регулярного выражения в заданной строке, как в следующем примере.

```
var pattern = /Java/g;
var text = "JavaScript лучше, чем Java!";
var result;
while((result = pattern.exec(text)) != null) {
  alert("Соответствие '" + result[0] + "'" +
    " в позиции " + result.index +
      поиск с позиции " + pattern.lastIndex);
}
```

Метод test() класса RegExp намного проще, чем exec(). Он принимает строку и возвращает true, если находит соответствие.

```
var pattern = /java/i;
pattern.test("JavaScript"); // => true
```

Вызов test() эквивалентен вызову exec() и возврату true, если возвращаемое значение метода exec() не равно null. Благодаря этой эквивалентности метод test() обрабатывает глобальное регулярное выражение так же, как и метод exec(): он начинает поиск в заданной строке с позиции, заданной значением свойства lastIndex, и, найдя соответствие, присваивает свойству lastIndex позицию символа, находящегося непосредственно после соответствия. Следовательно, с помощью метода test() можно проходить в цикле по строке так же, как и с помощью метода exec().

JavaScript на стороне клиента

В предыдущих главах рассматривался синтаксис JavaScript. Теперь же мы перейдем к использованию JavaScript в браузерах на стороне клиента. Большинство встречавшихся ранее примеров представляли собой корректный код JavaScript, но не привязанный к какому-либо контексту. Это были всего лишь фрагменты кода, выполняемые не в рабочей среде. В данной главе рассматривается рабочая среда на стороне клиента.

Внедрение JavaScript-кода в HTML-документ

Код JavaScript может находиться в HTML-документе между дескрипторами `<script>` и `</script>`.

```
<script>
  // Здесь находится код JavaScript
</script>
```

В листинге 10.1 приведен HTML-файл, содержащий простую JavaScript-программу. Комментарии объясняют, что делает программа, но главное назначение данного примера — не технические подробности, а демонстрация внедрения кода JavaScript в HTML-файл с таблицей CSS.

Листинг 10.1. Простые цифровые часы на основе JavaScript

```
<!DOCTYPE html> <!-- Это файл HTML5 -->
<html>          <!-- Корневой элемент -->
<head>          <!-- Заголовок -->
<title>Цифровые часы</title>
```

```
<script>        // Сценарий JavaScript
// Функция, выводящая текущее время
function displayTime() {
   var now = new Date(); // Получение времени
   // Поиск элемента с id="clock"
   var elt = document.getElementById("clock");
   // Вывод времени
   elt.innerHTML = now.toLocaleTimeString();
   // Повторение один раз в секунду
   setTimeout(displayTime, 1000);
}
// Запуск часов при загрузке HTML-документа
window.onload = displayTime;
</script>
<style>    /* Таблица CSS для часов */
#clock {   /* Стили для элемента id="clock" */
   font: bold 24pt sans;     /* Крупный шрифт */
   background: #ddf;         /* Серый фон */
   padding: 10px;            /* Отступ */
   border: solid black 2px; /* Черная рамка */
   border-radius: 10px;      /* Скругленные углы */
}
</style>
</head>
<body>                    <!-- Содержимое страницы -->
<h1>Цифровые часы</h1>    <!-- Заголовок -->
<span id="clock"></span>  <!-- Текущее время -->
</body>
</html>
```

В дескриптор `<script>` можно включить атрибут `src`, задающий адрес файла с кодом JavaScript.

```
<script src="../../scripts/util.js"></script>
```

Файл JavaScript должен содержать только код JavaScript — без дескрипторов `<script>` и других элементов HTML. Согласно общепринятым соглашениям файлы JavaScript имеют расширение `.js`.

Добавление в HTML-код дескриптора <script> с атрибутом src эквивалентно размещению содержимого указанного файла .js в этом же месте между дескрипторами <script> и </script>. Обратите внимание на то, что закрывающий дескриптор </script> в HTML-документе обязателен, даже если задан атрибут src и содержимого между дескрипторами <script> и </script> нет.

Язык JavaScript всегда применялся для создания сценариев на веб-страницах, поэтому часть его названия перешла в название дескриптора <script>. Элемент <script> имеет атрибут type, которому по умолчанию присвоено значение text/javascript. Можно присвоить его явно, но это необязательно.

Программирование на основе событий

Клиентские программы на JavaScript обычно работают асинхронно и управляются событиями. Когда загружается веб-страница, сценарии обычно инициализируют некоторые переменные и регистрируют функции обработки событий. Затем эти функции вызываются браузером при наступлении событий, для которых они зарегистрированы. Например, веб-приложение, в котором для каких-либо операций используются "горячие" клавиши, регистрирует события клавиатуры. События используются даже в неинтерактивных программах. Предположим, нужно создать программу, которая анализирует структуру документа и автоматически генерирует оглавление. В этом случае события взаимодействия с пользователем не нужны, но программа все же регистрирует обработчик события onload, чтобы программа определила, когда закончилась загрузка страницы и документ готов к автоматической генерации оглавления.

События и обработчики событий подробнее рассматриваются в главе 12.

Объект окна

Объект `Window` — главная точка входа во все клиентские средства JavaScript и функции API. Этот объект представляет окно или фрейм браузера, и на него можно ссылаться с помощью идентификатора `window`. В объекте `Window` определено свойство `location`, которое ссылается на объект `Location`, содержащий URL-адрес отображаемой веб-страницы. Свойство `location` позволяет сценариям загружать в окно браузера другие веб-страницы.

```
window.location = "http://www.oreilly.com/";
```

В объекте `Window` определены также метод `alert()`, который отображает диалоговое окно сообщений, и метод `setTimeout()`, который регистрирует функцию, автоматически вызываемую через заданный интервал времени. Выполнение приведенной ниже инструкции приведет к появлению сообщения `Привет!` через две секунды после вызова функции.

```
setTimeout(function() {alert("Привет!");}, 2000);
```

Обратите внимание на то, что в приведенном выше коде нет явного использования свойства `window`. Это необязательно, потому что на стороне клиента объект `Window` всегда является глобальным. Следовательно, объект `Window` находится на вершине цепочки областей видимости, и его свойства и методы являются глобальными переменными и функциями. У объекта `Window` есть свойство `window`, которое ссылается на этот же объект. Свойство `window` можно явно использовать в программе, но для обращения к свойствам глобального объекта окна это необязательно.

Как и в глобальном объекте, в объекте Window определен ряд свойств и методов, полезных для создания программ JavaScript. Наиболее важное свойство document подробно рассматривается в главе 11. Другие свойства и методы объекта Window рассматриваются в данной главе.

Таймеры

Встроенные методы setTimeout() и setInterval() позволяют зарегистрировать пользовательскую функцию, вызываемую один раз или многократно по истечении заданного интервала времени. Это важные глобальные функции на стороне клиента, поэтому они определены как методы объекта Window. В то же время эти функции предназначены для решения любых задач, необязательно связанных с окном.

Метод setTimeout() объекта Window регистрирует функцию, выполняемую через интервал времени, заданный в миллисекундах. Значение, возвращаемое методом setTimeout(), можно передать методу clearTimeout() для отмены выполнения зарегистрированной функции. Если вызвать метод setTimeout() и передать аргумент 0, зарегистрированная функция не будет вызвана немедленно. Она будет поставлена в очередь и вызвана в момент, когда завершится выполнение любого текущего обработчика события.

Метод setInterval() напоминает метод setTimeout() за исключением того, что регистрируемая функция вызывается многократно с интервалом, заданным в миллисекундах.

```
// Метод updateClock() вызывается каждые 60 секунд
setInterval(updateClock, 60000);
```

Как и setTimeout(), метод setInterval() возвращает значение, которое можно передать методу clearInterval() для отмены вызовов зарегистрированной функции.

Свойство location

Свойство location объекта Window ссылается на объект Location, который представляет URL-адрес документа, отображенного в окне. Кроме того, свойство location можно использовать для загрузки в окно другого документа.

Свойство href объекта Location содержит полный текст URL-строки текущего документа. Метод toString() объекта Location возвращает значение свойства href, поэтому обычно вместо location.href можно писать location.

Другие свойства объекта Location (protocol, host, hostname, port, pathname, search и hash) содержат отдельные части URL-адреса. Иногда их называют свойствами *декомпозиции* адреса. Кроме того, они поддерживаются объектом Link, создаваемым элементами <a> и <area> HTML-документа.

В объекте Location определен метод reload(), который заставляет браузер перезагрузить документ.

Объект Location можно использовать для перехода к другой странице. Для этого достаточно присвоить новый URL-адрес непосредственно свойству location.

```
location = "http://www.oreilly.com";
```

Свойству location можно также присвоить относительный URL-адрес. Он будет разрешен относительно текущего URL-адреса.

```
location = "page2.html"; // Следующая страница
```

Если в качестве относительного URL-адреса присвоить свойству location идентификатор фрагмента страницы, то браузер не загрузит новый документ, а перейдет к указанному фрагменту текущего документа. Специальный случай — идентификатор #top. Если ни один элемент до-

кумента не имеет атрибута id со значением top, то браузер перейдет к началу документа.

```
location = "#top";
```

Свойства декомпозиции URL-адреса объекта Location доступны для записи. При изменении любого из них немедленно обновляется свойство location, что заставляет браузер загрузить новый документ (или, в случае свойства hash, перейти в другое место текущего документа).

```
location.search = "?page=" + (pagenum+1);
```

История браузера

Свойство history объекта Window ссылается на объект History окна. В объекте History представлена история просмотра страниц в виде списка документов и состояний документов.

В объекте History есть методы back() и forward(), которые имитируют щелчки на кнопках Back (Назад) и Forward (Вперед) браузера. При вызове этих методов браузер переходит на предыдущую или следующую страницу. Метод go() принимает целое число, задающее переход на несколько страниц вперед или назад (в зависимости от знака аргумента). Приведенная ниже инструкция задает переход на две страницы назад.

```
history.go(-2);
```

Если объект Window содержит дочерние окна (например, элемент <iframe>, который рассматривается в главе 11), истории дочерних окон хронологически встраиваются в историю главного окна. Например, если выполнялись переходы в дочернем окне, то при вызове метода history.back() браузер может перейти к предыдущей стра-

нице дочернего окна, оставив текущую страницу главного окна без изменений.

Современные веб-приложения могут динамически изменять свое содержимое без загрузки нового документа. Для таких приложений часто необходимо, чтобы пользователь мог с помощью кнопок Back и Forward браузера переходить между состояниями, динамически создаваемыми приложением. Один из способов решения этой задачи состоит в сохранении строки, отображающей текущее состояние приложения, в свойстве `location.hash`. Новый документ при этом не загружается, но создается новый пункт истории, к которому пользователь позже может вернуться с помощью кнопки Back. В приложении для этого используется событие `hashchange`, генерируемое браузером. Приложение, которому нужно отслеживать кнопки Back и Forward, может зарегистрировать обработчик `window.onhashchange`.

Другой, более сложный способ управления историей браузера веб-приложением основан на использовании метода `history.pushState()` и соответствующего ему обработчика события `window.onpopstate`. В данной книге этот способ не рассматривается.

Информация о браузере и экране

Иногда сценарию необходима информация о браузере, в котором он выполняется, и о настольном компьютере, на котором установлен браузер. В данном разделе рассматриваются свойства `navigatior` и `screen` объекта `Windows`, ссылающиеся на объекты `Navigator` и `Screen`. Эти объекты предоставляют программе информацию, позволяющую настроить поведение сценария на основе характеристик среды. Объект `Navigator` содержит информацию о типе браузера и номере версии. Назван он так потому, что в далеком

прошлом он поддерживал только один браузер — Netscape Navigator, однако сейчас объект Navigator, несмотря на название, поддерживает все типы браузеров.

У объекта Navigator есть четыре свойства, содержащих информацию о текущем браузере.

- **appName** — полное имя браузера. Для IE это свойство равно Microsoft Internet Explorer, для Firefox — Netscape. В целях совместимости многие другие браузеры также возвращают строку Netscape.
- **appVersion**. Это свойство обычно начинается с числа, за которым следует подробная информация о производителе и версии браузера. Число, указанное в начале, обычно равно 4.0 или 5.0 и означает совместимость с браузерами четвертого или пятого поколения. Для строки appVersion нет стандартного формата, поэтому ее синтаксический анализ независимо от типа браузера невозможен.
- **n userAgent**. Эту строку браузер передает HTTP-серверу в заголовке USER-AGENT сообщения GET или POST. Обычно это свойство содержит всю информацию из appVersion и некоторые дополнительные подробности. Как и в случае appVersion, формат данного свойства не стандартизован.
- **platform**. Строка, идентифицирующая операционную систему и, возможно, оборудование, на котором выполняется браузер.

Кроме информации о браузере и версии, объект Navigator содержит ряд дополнительных свойств и методов. Ниже приведены некоторые стандартизованные и нестандартизованные свойства, часто использующиеся в сценариях JavaScript.

- **onLine**. Свойство navigator.onLine (если оно существует) сообщает, подключен ли в данный момент браузер к сети.
- **geolocation**. Объект Geolocation определяет функции, предназначенные для вычисления текущих географических координат браузера. Подробности работы с функциями объекта Geolocation в данной книге не рассматриваются.

Свойство screen объекта Window ссылается на объект Screen, предоставляющий информацию о размерах пользовательского монитора. Свойства width (ширина) и height (высота) содержат размеры экрана в пикселях. Свойства availWidth и availHeight содержат размеры доступной области экрана, в которую не входят некоторые элементы, такие как панель задач. Объект Screen часто используется для выяснения того, выполняется ли приложение на портативном устройстве, таком как планшет или мобильный телефон.

Диалоговые окна

Объект Window предоставляет три метода для отображения простых диалоговых окон. Метод alert() выводит сообщение с кнопкой OK и ожидает, пока пользователь щелкнет на ней. Метод confirm() требует подтверждения. Он выводит сообщение с кнопками OK и Отмена и ожидает, пока пользователь подтвердит или отменит некоторую операцию, щелкнув на одной из кнопок. Возвращаемое значение (true или false) зависит от того, на какой кнопке щелкнул пользователь. Метод prompt() выводит сообщение и текстовое поле, в которое пользователь может ввести строку, и возвращает эту строку. В приведенном ниже коде в качестве примера используются все три метода.

```
do {
  // Запрос строки
  var n = prompt("Введите свое имя");
  // Запрос подтверждения
  var ok = confirm("Имя " + n +   правильное?");
} while(!ok)
alert("Здравствуйте,   + n); // Вывод приветствия
```

Методы `alert()`, `confirm()` и `prompt()` легко использовать, однако из эстетических соображений на практике их применяют редко. На веб-страницах в Интернете они почти не встречаются.

Элементы документа как свойства окна

Если с помощью атрибута `id` присвоить элементу HTML-документа имя, которого нет в объекте `Window`, то этот объект будет иметь неперечислимое свойство с указанными именем и значением `HTMLElement`, представляющим элемент документа.

На стороне клиента объект `Window` играет роль глобального объекта, поэтому значение атрибута `id`, используемого в HTML-документе, становится глобальной переменной (если в программе еще нет глобальной переменной с этим же именем), доступной для сценария. Например, если в документе есть элемент `<button id="okay"/>`, то на данный элемент можно ссылаться в сценарии с помощью глобальной переменной `okay`.

Неявное использование атрибута `id` в качестве глобальной переменной — историческая традиция в развитии веб-браузеров. Данное средство необходимо для обратной совместимости с существующими веб-страницами, но в настоящее время использовать его не рекомендуется. Лучше явно задавать элементы с помощью методик, рассмотренных в главе 11.

Множественные окна и фреймы

Одно окно браузера настольного компьютера может содержать несколько вкладок, каждая из которых служит независимым *контекстом просмотра* (browsing context). Каждый контекст имеет собственный объект `Window` и жестко изолирован от всех остальных контекстов. Сценарии, выполняющиеся в одном контексте, обычно ничего не знают о существовании других контекстов и не могут взаимодействовать с их объектами `Window` или манипулировать содержимым их документов. В браузере, который не поддерживает вкладки или в котором отключены вкладки, может быть открыто одновременно несколько окон. В этом случае каждое окно также имеет собственный объект `Window` и контекст просмотра, изолированные от других окон и не зависящие от них.

HTML-документ может содержать вложенные документы, определяемые элементом `<iframe>`. Элемент `<iframe>` создает вложенный контекст просмотра, представленный отдельным объектом `Window`. Устаревшие элементы `<frameset>` и `<frame>` также создают вложенные контексты просмотра, и каждый элемент `<frame>` представлен объектом `Window`. Клиентский код JavaScript почти не отличает вкладки, окна и фреймы друг от друга; для кода все они — контексты просмотра, представленные объектами `Window`. Вложенные контексты просмотра не изолированы друг от друга, как отдельные вкладки. Сценарий, выполняющийся во фрейме, всегда может видеть родительские и дочерние контексты, хотя правило ограничения домена (подробнее об этом — далее) может запретить сценарию просматривать документы в других фреймах. Вложенные фреймы подробнее рассматриваются в главе 11.

Объект `Window` является глобальным на стороне клиента, поэтому каждое окно или фрейм имеет отдельный

контекст просмотра. Тем не менее код JavaScript одного окна может (с учетом правила ограничения домена) обращаться к объектам, свойствам и методам, определенным в другом окне.

Связи между фреймами

Вы уже знаете, что код JavaScript в любом окне или фрейме может ссылаться на собственный объект `Window` по имени `window` или `self`. Фрейм может ссылаться на объект `Window` родительского окна или фрейма с помощью свойства `parent`.

```
parent.history.back();
```

Объект `Window`, представляющий окно или вкладку верхнего уровня, не имеет контейнера, поэтому свойство `parent` ссылается на собственное окно.

```
parent == self; // Для верхнеуровневых окон
```

Если фрейм находится в другом фрейме, который сам находится в верхнеуровневом окне, то первый фрейм может ссылаться на верхнеуровневое окно посредством выражения `parent.parent`. Для этого же можно использовать свойство `top`. Независимо от того, как глубоко вложен фрейм, его свойство `top` всегда ссылается на родительское верхнеуровневое окно. Если объект `Window` представляет верхнеуровневое окно, то свойство `top` ссылается на собственное окно. Во фрейме, являющемся непосредственным потомком верхнеуровневого окна, свойства `top` и `parent` содержат одно и то же значение.

Свойства `parent` и `top` позволяют сценарию обращаться к предкам фрейма. Кроме того, существует несколько способов обращения к потомкам фрейма или окна. Фрейм создается с помощью элемента `<iframe>`. В коде можно получить объект `Element`, представляющий элемент `<iframe>`

так же, как и любой другой элемент. Предположим, в документе есть дескриптор <iframe id="f1">. Тогда получить объект Element, представляющий данный фрейм, можно следующим образом:

```
var e = document.getElementById("f1");
```

Элемент <iframe> имеет свойство contentWindow, ссылающееся на объект Window фрейма, поэтому объект Window фрейма можно получить следующим образом:

```
var kid = document.getElementById("f1").contentWindow;
```

Можно пойти обратным путем: от объекта Window, представляющего фрейм, к элементу Element, содержащему фрейм. Это делается с помощью свойства frameElement объекта Window. У объекта Window, представляющего верхнеуровневое окно, а не фрейм, свойство frameElement равно null.

```
var elt = document.getElementById("f1");
var w = elt.contentWindow;
w.frameElement === elt  // Для фрейма всегда true
w.frameElement === null // Верхнеуровневое окно
```

Однако для получения ссылки на дочерний фрейм окна обычно можно обойтись без метода getElementByID() и свойства contentWindow. Каждый объект Window имеет свойство frames, ссылающееся на дочерние фреймы в данном окне или фрейме. Свойство frames содержит массивоподобный объект, который можно индексировать численно или по именам фреймов. Для ссылки на первый дочерний фрейм окна можно использовать выражение frames[0]. Для ссылки на третий дочерний элемент второго вложенного дочернего фрейма напишите frames[1].frames[2]. Код, выполняющийся во фрейме, может ссылаться на "родственный" фрейм следующим образом: parent.frames[1]. Обратите внимание на то, что элементы

массива frames[] являются объектами Windows, а не элементами <iframe>.

Если в элементе <iframe> задан атрибут name или id, то фрейм можно индексировать по имени и номеру. Например, на фрейм с именем f1 можно сослаться так: frames["f1"] или frames.f1.

Атрибут name или id элемента <iframe> можно применить для присвоения фрейму имени, которое можно использовать в коде JavaScript. Однако если применить атрибут name, то заданное в нем имя становится также значением свойства name объекта Window, представляющего фрейм. Имя, заданное таким способом, может служить значением атрибута target гиперссылки на фрейм.

Применение JavaScript во взаимодействующих окнах

Каждое окно или фрейм имеет собственный контекст просмотра и глобальный объект Window. Однако если код одного окна или фрейма может ссылаться на другое окно или фрейм (и правило ограничения домена не препятствует этому), то сценарии одного окна или фрейма могут взаимодействовать со сценариями других окон.

Предположим, на веб-странице есть два элемента <iframe> с именами A и B, и эти два фрейма содержат документы с одного общего сервера, а документы содержат взаимодействующие сценарии. В сценарии фрейма A можно определить переменную i.

```
var i = 3;
```

Эта переменная является свойством глобального объекта, т.е. объекта Window. Код фрейма A может ссылаться на нее с помощью идентификатора i или посредством явной ссылки на объект окна.

```
window.i
```

Сценарий фрейма B может ссылаться на объект Window фрейма A, поэтому он может ссылаться также на свойства этого объекта, в частности на свойство i.

```
parent.A.i = 4;
```

Как вы помните, ключевое слово function, определяющее функцию, одновременно создает переменную, как и ключевое слово var. Если сценарий во фрейме B объявляет функцию f, то она становится глобальной переменной во фрейме B, и код фрейма B может вызывать ее с помощью выражения f(). Однако код фрейма A должен сослаться на функцию f как на свойство объекта Window фрейма B.

```
parent.B.f();
```

Если в коде фрейма A эта функция используется часто, можно присвоить ее переменной фрейма A, чтобы ее удобнее было вызывать. Можно даже присвоить ей это же имя.

```
var f = parent.B.f;
```

Теперь код фрейма A может вызывать функцию f() так же, как код фрейма B.

Сделав таким образом функцию общей для нескольких фреймов или окон, важно учитывать правила лексической области видимости: функция выполняется в области видимости, в которой она была определена, а не вызвана. Следовательно, если функция f() ссылается на глобальные переменные, интерпретатор браузера ищет эти переменные во фрейме B, даже если функция была вызвана во фрейме A.

Правило ограничения домена

Так называется политика ограничений безопасности, накладываемых на код JavaScript, который взаимодей-

ствует с веб-страницами. Обычно правило ограничения домена вступает в силу, когда на веб-страницах появляются элементы <iframe>. В этом случае политика определяет правила взаимодействия кода JavaScript в одном окне или фрейме с содержимым других окон или фреймов. В частности, сценарий может читать свойства только тех окон и документов, которые имеют то же происхождение, что и документ, содержащий сценарий.

Происхождение документа определяется в виде протокола, хоста и порта в URL-адресе, из которого загружен документ. Документы, загруженные с других веб-серверов, а также с других портов этого же хоста, имеют другое происхождение. Документы, загруженные с помощью протоколов `http:` и `https:`, имеют разное происхождение, даже если они загружались с одного веб-сервера.

Важно понимать, что происхождение сценария не влияет на политику. На нее влияет только происхождение документа, в который внедрен сценарий. Предположим, например, что сценарий, обслуживаемый хостом `A`, добавлен (с помощью свойства `src` или элемента <script>) на веб-страницу, обслуживаемую хостом `B`. Тогда сценарий происходит от хоста `B` и обладает правом доступа к содержимому документов хоста `B`. Если сценарий создает фрейм и загружает второй документ хоста `B`, то он имеет право доступа и ко второму документу. Но если сценарий открывает еще один фрейм и загружает в него документ из хоста `C` (или даже из хоста `A`), то правило ограничения домена запретит доступ сценария к этому документу.

Правило ограничения домена применяется не ко всем свойствам всех объектов окна другого происхождения. Тем не менее оно применяется ко многим из них, в частности, ко всем свойствам объекта `Document`. Необходимо учитывать, что любое окно или фрейм, содержащий документ с другого

сервера, недоступен для сценария. Правило ограничения домена применяется также к HTTP-запросам, генерируемым объектом `XMLHttpRequest` (подробнее об этом — в главе 13). Этот объект позволяет коду JavaScript на стороне клиента передавать любые HTTP-запросы к веб-серверу, с которого был загружен хостирующий документ, но он не позволяет сценарию взаимодействовать с другими веб-серверами.

Работа с документами

Код JavaScript на стороне клиента необходим для того, чтобы превратить статические HTML-документы в интерактивные веб-приложения. Содержимое окна браузера представлено объектом `Document`, который является предметом рассмотрения данной главы. Объект `Document` — это центральная часть большой библиотеки API-функций, называемой DOM (Document Object Model — объектная модель документа) и предназначенной для программного манипулирования содержимым документов.

Обзор модели DOM

Структура модели DOM не очень сложная, но в ней есть ряд архитектурных особенностей, которые нужно понимать. Вложенные элементы HTML или XML представлены в DOM в виде дерева объектов. Древовидное представление документа содержит узлы, соответствующие дескрипторам или элементам, таким как `<body>` или `<p>`, и узлы, соответствующие текстовым строкам. Представление документа может также содержать узлы, соответствующие комментариям. Рассмотрим следующий простой документ.

```
<html>
  <head>
    <title>Образец документа</title>
  </head>
  <body>
    <h1>HTML-документ</h1>
```

```
    <p>Это <i>простой</i> документ.</p>
</html>
```

Представление DOM данного документа имеет древовидную структуру, показанную на рис. 11.1.

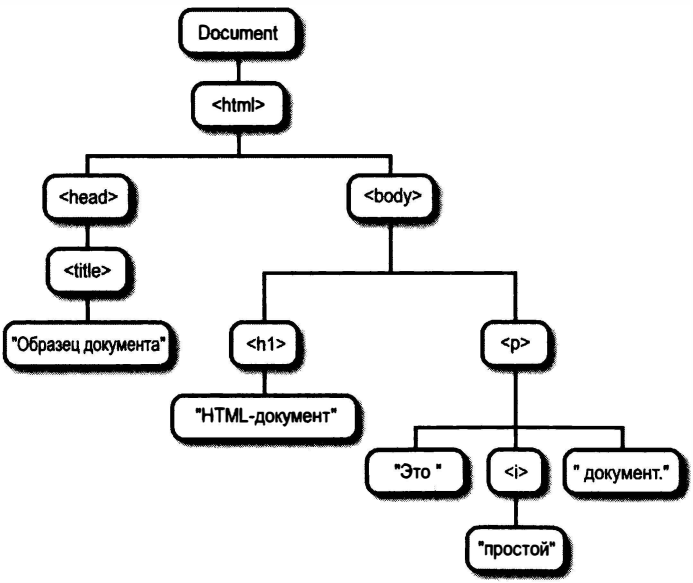


Рис. 11.1. Древовидное представление HTML-документа

Если вы не встречались с древовидными структурами, вам полезно будет ознакомиться со специальной терминологией (пусть вас не смущает то, что корень дерева находится вверху, а ветви растут вниз). Узел, расположенный непосредственно над данным узлом, называется *родительским* по отношению к нему. Узел, расположенный непосредственно под данным узлом, называется *дочерним*.

Два узла на одном и том же уровне, имеющие общий родительский узел, называются *братьями*. Узлы, расположенные ниже данного узла и связанные с ним, называются *потомками*, а расположенные выше — *предками* данного узла.

Каждый прямоугольник на рис. 11.1 — это узел документа, представленный объектом `Node`. В следующих разделах рассматриваются свойства и методы объекта `Node`. Обратите внимание на то, что на рис. 11.1 есть три разных типа узлов. В корне дерева находится узел `Document`, представляющий весь документ. Элементы HTML представлены узлами типа `Element`, а текстовое содержимое элементов — узлами типа `Text`. Классы `Document`, `Element` и `Text` являются подклассами класса `Node`. В модели DOM наиболее важны классы `Document` и `Element`, и данная глава посвящена главным образом их свойствам и методам.

Типы и подтипы узлов образуют иерархию, показанную на рис. 11.2. Обратите внимание на формальное различие между типами `Document` и `Element` с одной стороны и типами `HTMLDocument` и `HTMLElement` — с другой. Тип `Document` представляет документ HTML или XML, а тип `Element` — элемент документа. Эти типы общие для всех элементов и документов. В то же время типы `HTMLDocument` и `HTMLElement` специфичны для документов и элементов HTML. В данной книге часто используются обобщенные имена классов `Document` и `Element` для ссылки на документы и элементы HTML.

Тип `HTMLElement` имеет много подтипов, представляющих конкретные элементы HTML. В каждом из этих подтипов определены свойства JavaScript, дублирующие HTML-атрибуты конкретного элемента или группы элементов. В некоторых из классов, специфичных для элементов, определены дополнительные свойства и методы, отсутствующие в синтаксисе HTML.

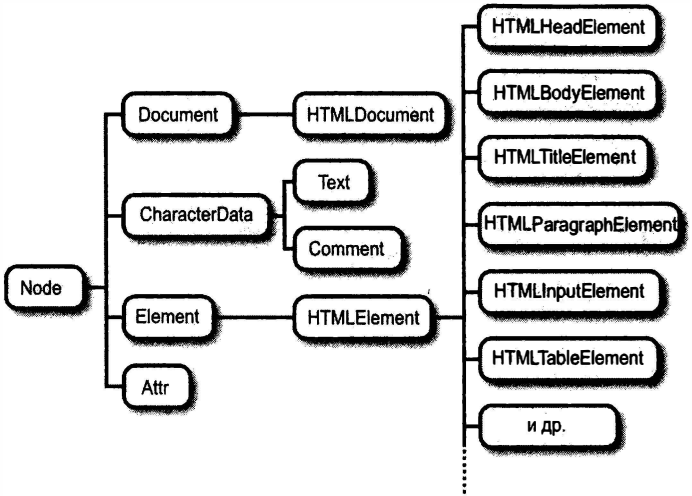


Рис. 11.2. Часть иерархии узлов документа

Выбор элементов документа

Обычно программа JavaScript на стороне клиента решает свои задачи путем манипулирования одним или несколькими элементами документа. При запуске программы в ней можно использовать глобальную переменную `document` для ссылки на объект `Document`. Для манипулирования конкретным элементом документа необходимо каким-либо образом *выбрать* объект `Element`, ссылающийся на нужный элемент. В модели DOM предусмотрен ряд способов выбора элемента:

- по значению атрибута `id`;
- по значению атрибута `name`;
- по имени дескриптора;

- по классу или классам CSS;
- по соответствию заданному селектору CSS.

В следующих разделах рассматривается каждый способ.

Выбор элементов по идентификатору

Каждый элемент HTML имеет атрибут `id`. Значение этого атрибута должно быть уникальным во всем документе, т.е. никакие два элемента одного документа не могут иметь одно и то же значение `id`. Выбрать элемент на основе уникального значения `id` можно с помощью метода `getElementById()` объекта `Document`.

```
var sect1 = document.getElementById("section1");
```

Это наиболее простой и распространенный способ выбора элемента. Если сценарий должен работать с некоторым набором элементов документа, присвойте атрибутам `id` этих элементов определенные значения. Если нужно выбрать более одного элемента по идентификатору, то полезной может быть функция `getElements()`, приведенная в листинге 11.1.

Листинг 11.1. Выбор многих элементов по значениям `id`

```
/*
* Эта функция ожидает произвольное количество
* строковых аргументов. Каждый аргумент она
* интерпретирует как идентификатор элемента и вызывает
* метод document.getElementById() для каждого элемента.
* Функция возвращает объект, связывающий
* значения id с объектами Element
*/
function getElements(/*значения id...*/) {
  var elements = {}; // Начинаем с пустого объекта
  for(var i = 0; i < arguments.length; i++) {
    var id = arguments[i]; // Идентификатор элемента
    var elt = document.getElementById(id);
```

```
    if (elt == null)
       throw new Error("Такого элемента нет:   + id);
    elements[id] = elt;     // Привязка к элементу
  }
  return elements; // Возвращение объекта
}
```

Выбор элементов по имени

Атрибут `name` изначально был предназначен для присвоения имен элементам форм. Значение атрибута `name` используется при передаче данных формы на сервер. Как и атрибут `id`, атрибут `name` используется для присвоения имени элементу. Однако в отличие от `id`, значение атрибута `name` не обязательно должно быть уникальным. Многие элементы могут называться одинаково. Эта особенность часто применяется в переключателях и флажках форм. Кроме того, в отличие от `id`, атрибут `name` допустим только в подмножестве элементов HTML, в которое входят формы, элементы форм и элементы `<iframe>` и `<img>`.

Для выбора элементов HTML на основе значений атрибута `name` необходимо вызвать метод `getElementsByName()` через объект `Document`.

```
var btns = document.getElementsByName("color");
```

Метод `getElementsByName()` возвращает объект `NodeList`, который работает как массив объектов `Element`, доступный только для чтения.

Установка атрибута `name` элемента `<form>`, `<img>` или `<iframe>` создает свойство объекта `Document` с именем, равным значению атрибута (если, конечно, у объекта `Document` на тот момент не было свойства с этим именем). Если данное имя присвоено только одному элементу, то значением свойства объекта `Document` становится сам элемент. Если данное имя присвоено нескольким элементам, то значени-

ем свойства с данным именем становится объект NodeList, содержащий массив элементов. Однако важно учитывать, что свойства документа, созданные для именованных элементов <iframe>, ссылаются на объект Window фрейма, а не на объект Element.

Все это означает, что элементы можно выбирать по имени посредством свойств объекта Document, имеющих соответствующие имена.

```
// Получение элемента
// <form name="shipping_address">
var form = document.shipping_address;
```

Выбор элементов по типу дескриптора

Выбирать элементы можно также по их типу (иногда пишут "по имени дескриптора") с помощью метода getElementsByTagName() объекта Document. Например, получить массивоподобный (см. главу 6) объект, содержащий объекты Element всех элементов , присутствующих в документе, можно с помощью следующей инструкции:

```
var spans = document.getElementsByTagName("span");
```

Как и getElementsByName(), метод getElementsByTagName() возвращает объект NodeList. Элементы возвращаемого объекта NodeList расположены в том же порядке, что и в документе. Например, первый элемент <p> можно получить следующим образом:

```
var firstpara = document.getElementsByTagName("p")[0];
```

Дескрипторы HTML нечувствительны к регистру, поэтому при использовании метода getElementsByTagName() с документом HTML выполняется сравнение имен, нечувствительное к регистру букв. Следовательно, приведенная выше инструкция вернет не только элементы , но и элементы <SPAN>.

Передав методу getElementsByTagName() метасимвол "*", можно получить объект NodeList, содержащий все элементы документа.

Метод getElementsByTagName() определен не только в классе Document, но и в классе Element. Он работает так же, как и в классе Document, но выбирает только элементы, являющиеся потомками элемента, через который вызван метод. Например, чтобы найти все элементы , находящиеся в первом элементе <p>, необходимо выполнить следующие инструкции.

```
var firstp= document.getElementsByTagName("p")[0];
var firstpSpans = firstp.getElementsByTagName("span");
```

По историческим причинам свойства быстрого доступа к узлам некоторых типов определены в классе HTMLDocument. Например, свойства images, forms и links ссылаются на объекты, ведущие себя как доступные для чтения массивы элементов <img>, <form> и <a> (но только тех элементов <a>, которые имеют атрибут href). Эти свойства ссылаются на объекты HTMLCollection, напоминающие объекты NodeList, но отличающиеся от них тем, что их можно индексировать по идентификаторам или именам. Выше было показано, что на элемент <form> можно сослаться с помощью следующего выражения:

```
document.shipping_address
```

Свойство document.forms позволяет сослаться на элемент <form> с атрибутом id или name следующим образом:

```
document.forms.shipping_address;
```

В классе HTMLDocument определены два свойства (body и head), ссылающиеся на специальные одиночные элементы, а не на коллекции элементов. Свойство document.body ссылается на элемент <body>, а свойство document.head — на элемент <head>. Свойство documentElement класса Document

ссылается на корневой элемент документа. В HTML-документе корневым всегда является элемент <html>.

Классы NodeList и HTMLCollection

Методы getElementsByName() и getElementsByTagName() возвращают объекты NodeList, а такие свойства, как document.images и document.forms, — объекты HTMLCollection.

Это массивоподобные объекты, доступные только для чтения. У них есть свойство length. Кроме того, их можно индексировать (но только для чтения, а не для записи), как настоящие массивы. По содержимому объекта NodeList или HTMLCollection можно проходить в цикле.

```
// Сокрытие всех изображений
for(var i = 0; i < document.images.length; i++)
  document.images[i].style.display = "none";
```

Одна из наиболее важных и удивительных особенностей объектов NodeList и HTMLCollection состоит в том, что они не являются статическими снимками состояний документа. Они постоянно обновляются при каждом изменении документа. Предположим, функция getElementsByTagName('div') вызывается в документе, в котором нет ни одного элемента . Если после этого вставить в документ новый элемент , то он автоматически станет членом полученного объекта NodeList, а свойство length увеличится на единицу.

Выбор элементов по классам CSS

Атрибут class элемента HTML содержит список (возможно, пустой) идентификаторов, разделенных пробелами. Каждый идентификатор присваивает элементу класс CSS. Любой элемент, имеющий этот же идентификатор в атрибуте class, является частью того же набора

элементов. В синтаксисе JavaScript ключевое слово class зарезервировано для других целей, поэтому для хранения значения атрибута class используется свойство className. Атрибут class обычно используется совместно с таблицей CSS для присвоения презентационных стилей всем членам набора. Кроме того, в HTML5 определен метод getEmelementsByClassName(), позволяющий выбрать набор элементов на основе идентификаторов, приведенных в атрибуте class.

Как и getElementsByTagName(), метод getElementsByClassName() можно вызвать либо через документ, либо через элемент. В любом случае он вернет динамически обновляемый объект NodeList, содержащий все потомки документа или элемента данного класса. Метод getElementsByClassName() принимает один строковый аргумент, в котором, однако, можно задать список из многих идентификаторов, разделенных пробелами. Будут возвращены только элементы, в атрибуте class которых содержатся все идентификаторы классов, заданных при вызове. Последовательность идентификаторов не играет роли. Ниже приведены примеры использования метода getElementsByClassName().

```
// Поиск всех элементов класса warning
var w = document.getElementsByClassName("warning");
// Поиск потомков классов fatal и error
// элемента с идентификатором log
var log = document.getElementById("log");
var fatal = log.getElementsByClassName("fatal error");
```

Выбор элементов по селекторам CSS

Селекторы CSS описывают элементы и наборы элементов документа. В данной книге синтаксис селекторов CSS не рассматривается, приведено лишь несколько при-

меров. Элементы можно идентифицировать по значениям id, именам дескрипторов и классам.

```
#nav      // Любой элемент с id="nav"
div       // Любой элемент <div>
.warning // Любой элемент класса warning
```

В более общем случае элементы можно выбирать по значениям атрибутов.

```
p[lang="fr"] // Абзац на французском языке
*[name="x"]  // Элементы с атрибутом name="x"
```

Базовые селекторы можно объединять.

```
span.fatal.error  // элемент <span> классов fatal и error
span[lang="fr"].warning // предупреждение на французском
```

Селекторы могут также определять структуру документа.

```
#log span // Любой потомок типа <span>
          // элемента log
#log>span // Любой непосредственный потомок <span>
          // элемента log
body>h1:first-child // Первый непосредственный
                    // потомок <h1> элемента <body>
```

Селекторы можно объединять для выбора многих элементов или наборов элементов.

```
div, #log // Все элементы <div> и log
```

Селекторы CSS позволяют выбирать элементы любыми описанными выше способами: по идентификатору, по имени, по типу дескриптора и по имени класса. С помощью метода querySelectorAll() объекта Document можно выбрать элементы, соответствующие селектору CSS. Этот метод принимает строковый аргумент, содержащий селектор CSS, и возвращает объект NodeList, представляющий все элементы документа, соответствующие селектору. В отличие от описанных выше методов выбора

элементов, метод querySelectorAll() возвращает не динамический объект NodeList. Это означает, что полученный таким образом объект NodeList содержит элементы, соответствовавшие селектору на момент вызова и не обновляемые при изменении документа. Если ни один элемент не соответствует селектору, метод querySelectorAll() возвращает пустой объект NodeList. Если строка селекторов неправильная, метод querySelectorAll() генерирует исключение.

Кроме querySelectorAll(), в объекте Document определен метод querySelector(), который возвращает не все, а только первый (в последовательности документа) элемент, соответствующий селектору, или null, если соответствующий элемент не найден.

Эти же два метода определены и в объекте Element. При их вызове через объект Element, а не Document, поиск элементов, соответствующих селектору, выполняется во всем документе, а затем результирующий набор фильтруется таким образом, что возвращаются только потомки указанного элемента.

Структура и обход документа

После выбора элемента иногда необходимо найти структурно связанные с ним части документа, такие как родительские, братские или дочерние узлы. Объект Document и входящие в него объекты Element и Text являются объектами класса Node. В классе Node определены следующие важные свойства.

- **parentNode** — объект Node, родительский по отношению к данному (или null, если родительского нет).
- **childNodes** — массивоподобный объект NodeList, доступный только для чтения и динамически обнов-

ляемый при изменении документа. Содержит дочерние узлы.

- `firstChild` и `lastChild` — первый и последний дочерние узлы данного узла или `null`, если у данного узла нет непосредственных потомков.
- `nextSibling` и `previousSibling` — следующий и предыдущий братские узлы данного узла. Два узла называются братскими, если они имеют общий непосредственный родительский узел. Последовательность братских узлов в коде JavaScript совпадает с их последовательностью в документе. Эти свойства соединяют узлы в двойной связанный список.
- `nodeType` — тип узла. Для узла `Document` это свойство равно `9`, для узла `Element` — `1`, для узла `Text` — `3`, для узла `Comment` — `8`.
- `nodeValue` — текстовое содержимое узла `Text` или `Comment`.
- `nodeName` — имя дескриптора объекта `Element`, преобразованное в буквы верхнего регистра.

С помощью этих свойств объекта `Node` можно написать выражения, ссылающиеся на второй дочерний узел первого дочернего узла `Document`. Эти выражения равноправные.

```
document.childNodes[0].childNodes[1]
document.firstChild.firstChild.nextSibling
```

Рассмотрим следующий документ.

```
<html>
<head><title>Проверка</title></head>
<body>Привет!</body>
</html>
```

Элемент <body> является вторым дочерним узлом первого дочернего узла. Его свойство nodeType равно 1, а свойство nodeName — BODY.

Не забывайте, что рассматриваемый программный интерфейс очень чувствителен к изменениям документа. Если вставить в документ единственную строку между дескрипторами <html> и <head>, то узел Text, представляющий новую строку, станет первым дочерним элементом первого дочернего узла, а вторым дочерним элементом станет <head> вместо <body>.

Если нам нужны объекты Element, а не тексты и пробелы, то более полезным будет программный интерфейс, позволяющий трактовать документ как дерево объектов Element, игнорируя узлы Text и Comment, которые также являются частью документа.

Первая часть такого интерфейса на основе элементов — свойство children объекта Element. Как и childNodes, оно содержит объект NodeList, но, в отличие от свойства childNodes, список содержит только объекты Element.

Вторая часть интерфейса на основе элементов — перечисленные ниже свойства объекта Element, аналогичные дочерним и братским свойствам объекта Node.

- **firstElementChild** и **lastElementChild** — то же, что и firstChild и lastChild, но содержат только объекты Element.
- **nextElementSibling** и **previousElementSibling** — то же, что и nextSibling и previousSibling, но содержат только братские узлы Element.
- **childElementCount** — количество дочерних узлов Element. Возвращает то же значение, что и children.length.

Атрибуты

Элемент HTML состоит из имени дескриптора и набора пар "имя–значение", которые называются *атрибутами*. Например, элемент <a>, определяющий гиперссылку, имеет атрибут с именем href и значением, указывающим целевой адрес гиперссылки. Значения атрибутов доступны в коде JavaScript как свойства объекта HTMLElement, представляющего данный элемент в коде HTML. В типе HTMLElement определены универсальные атрибуты, такие как id, title, lang и dir, а также свойства, указывающие на обработчики событий, такие как onclick. В специальных типах определены атрибуты, специфичные для данного элемента. Например, для получения URL-адреса изображения можно использовать свойство src объекта HTMLElement, представляющего элемент <img>.

```
var img = document.getElementById("myimage");
var url = img.src; // Атрибут src
img.id = "myimg"   // Изменение атрибута id
```

Аналогично можно установить атрибуты передачи формы элемента <form> с помощью кода JavaScript.

```
var f = document.forms[0]; // Первая форма
f.method = "POST";         // Задание метода передачи
f.action = "http://www.example.com/submit.php";
```

Имена атрибутов HTML нечувствительны к регистру, а имена свойств JavaScript чувствительны. Чтобы преобразовать имя атрибута в имя свойства, запишите его в нижнем регистре. Если атрибут состоит из нескольких слов, то все слова, кроме первого, начинаются с буквы в верхнем регистре, например defaultChecked или tabIndex.

Некоторые имена атрибутов HTML совпадают с зарезервированными ключевыми словами JavaScript. Для них необходимо добавить к имени атрибута префикс html.

Например, атрибут for элемента <label> становится свойством htmlFor. В JavaScript слово class зарезервированное, но не используется, поэтому важный атрибут class является исключением из приведенного выше правила: в коде JavaScript соответствующее свойство называется className.

Свойства JavaScript, представляющие атрибуты HTML, обычно имеют строковые значения. Если атрибут булев или числовой (например, атрибуты defaultChecked и maxLength элемента <input>), то значение свойства не строковое, а булево или числовое. Атрибуты обработчиков событий всегда имеют в качестве значения объект Function (или null). В HTML5 определено несколько атрибутов (таких, как атрибут form элемента <input> и родственных элементов), преобразующих идентификатор элемента в фактический объект Element.

Как было показано выше, в типе HTMLElement и его подтипах определены свойства, соответствующие стандартным атрибутам элементов HTML. В типе Element определены также методы getAttribute() и setAttribute(), которые можно использовать для чтения и записи значений нестандартных атрибутов HTML.

```
var image = document.images[0];
var width = parseInt(image.getAttribute("WIDTH"));
image.setAttribute("class", "thumbnail");
```

В этом коде обратите внимание на два важных различия между используемыми в нем методами и описанным выше интерфейсом на основе свойств. Во-первых, значения атрибутов всегда интерпретируются как строки. Метод getAttribute() никогда не возвращает число, объект или булево значение. Во-вторых, в этих методах используются стандартные имена атрибутов, даже если они являются зарезервированными ключевыми словами

JavaScript. В элементах HTML имена атрибутов нечувствительны к регистру.

В объекте `Element` определены также два родственных метода, `hasAttribute()` и `removeAttribute()`, которые проверяют наличие именованного атрибута или удаляют его. Эти методы полезны при работе с булевыми атрибутами (такими, как атрибут `disabled` элемента формы), у которых играет роль их присутствие или отсутствие, но не их значение.

Содержимое элемента

Взгляните еще раз на рис. 11.1 и попытайтесь ответить на вопрос: "Что представляет собой содержимое элемента `<p>`?" На этот вопрос можно дать три ответа.

- Строка разметки: `Это <i>простой</i> документ.`
- Строка текста: `Это простой документ.`
- Набор текстовых узлов: левый узел `Text`, узел `Element`, его дочерний узел `Text` и правый узел `Text`.

Каждый из этих ответов правильный, и каждый может быть полезным в разных ситуациях. Они отражают три способа представления документа: в виде HTML-кода, в виде текста и в виде древовидной структуры.

Содержимое элемента в виде HTML-кода

Свойство `innerHTML` объекта `Element` возвращает содержимое элемента в виде строкуи разметки. Присвоение значения этому свойству приводит к запуску синтаксического анализатора браузера и замене текущего содержимого представлением новой строки, полученным в результате синтаксического разбора.

Процедура синтаксического анализа в браузерах хорошо оптимизирована, поэтому операция присваивания нового значения свойству `innerHTML` выполняется достаточно эффективно, даже несмотря на то, что для ее выполнения необходим синтаксический проход по тексту. Однако повторное выполнение данной операции с помощью оператора += снижает эффективность, потому что браузер отбрасывает предыдущий результат синтаксического разбора и вновь выполняет этапы сериализации, синтаксического разбора и визуализации.

Метод `insertAdjacentHTML()` позволяет вставлять строку с произвольной HTML-разметкой рядом с заданным элементом. Разметка передается методу через второй строковый аргумент, а значение слова "рядом" зависит от первого аргумента. Доступны четыре значения первого аргумента: `beforebegin` (перед началом), `afterbegin` (после начала), `beforeend` (перед концом) и `afterend` (после конца). Эти значения задают вставку в точки, показанные на рис. 11.3.

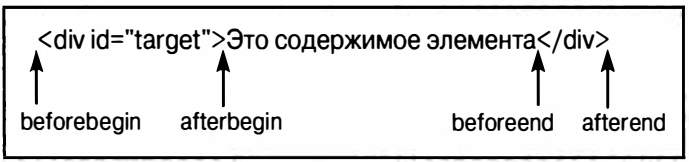


Рис. 11.3. Точки вставки при использовании метода `insertAdjacentHTML()`

Содержимое элемента в виде простого текста

Иногда необходимо получить содержимое элемента в виде простого неформатированного текста или вставить простой текст в документ, не заменяя специальные символы (амперсанд, угловые скобки и др.) управляющими последовательностями. Обычно это делают с помощью свойства `textContent` объекта `Node`.

```
// Получение первого элемента <p>
var para = document.getElementsByTagName("p")[0];
// Извлечение текста первого элемента <p>
var text = para.textContent;
// Изменение содержимого абзаца
para.textContent = "Привет!";
```

Содержимое элемента в виде набора узлов

Работать с содержимым элемента можно как со списком дочерних узлов, каждый из которых, в свою очередь, имеет собственный набор потомков. В этом случае обычно важны только узлы типа `Text`.

В листинге 11.2 показана пользовательская функция `textContent()`, которая рекурсивно проходит по дочерним элементам и объединяет текстовое содержимое всех потомков типа `Text`. Чтобы понять, как работает эта функция, вспомните, что в свойстве `nodeValue`, определенном в типе `Node`, находится содержимое узла `Text`.

Листинг 11.2. Поиск всех узлов `Text`, являющихся потомками заданного элемента

```
// Возвращает неформатированное текстовое
// содержимое элемента e, рекурсивно проходя по
// дочерним элементам. Эта функция работает
// аналогично свойству textContent
function textContent(e) {
  var c, type, s = "";
  for(c=e.firstChild; c!=null; c=c.nextSibling) {
    type = c.nodeType;
    if (type === 3)        // Текстовый узел:
      s += c.nodeValue;    // добавление текста
    else if (type === 1)   // Узел Element:
      s += textContent(c); // рекурсия
  }
  return s;
}
```

Обратите внимание на то, что свойство nodeValue доступно как для чтения, так и для записи. С его помощью можно изменять содержимое узла Text.

Создание, вставка и удаление узла

Вы уже знаете, как читать и обновлять содержимое документа путем манипулирования строками разметки и неформатированного текста. Проходя по документу с помощью синтаксического анализатора, можно просматривать отдельные узлы Element и Text, из которых состоит документ. Изменять документ можно также на уровне индивидуальных узлов. В типе Document определены методы создания объектов Element и Text, а в типе Node — методы вставки, удаления и замены узлов дерева. В приведенной ниже функции демонстрируются создание и вставка элемента в документ.

```
// Асинхронная загрузка и выполнение сценария
function loadasync(url) {
  // Создание элемента <script>
  var s = document.createElement("script");
  // Установка его атрибута src
  s.src = url;
  // Вставка элемента <script> в раздел <head>
  document.head.appendChild(s);
}
```

С помощью метода createElement() объекта Document в документе можно создать узел Element. В качестве аргумента методу нужно передать имя дескриптора.

Текстовые узлы создаются аналогичным методом.

```
var t = document.createTextNode("Это содержимое");
```

Еще один способ создания нового узла состоит в копировании существующего. Каждый узел имеет метод cloneNode(), который возвращает копию узла. Получив

аргумент true, он создаст рекурсивную копию всех потомков, а получив аргумент false, он создаст мелкую копию данного узла.

Созданный узел можно вставить в документ с помощью метода appendChild() или insertBefore() объекта Node. Метод appendChild() вызывается через узел Element, в который нужно вставить новый узел. Вставка выполняется в конец списка дочерних узлов, и новый узел получает статус lastChild.

Метод insertBefore() напоминает метод appendChild(), но в отличие от него получает не один, а два аргумента. Первый аргумент — вставляемый узел. Второй аргумент — узел, перед которым его нужно вставить. Метод insertBefore() вызывается через узел, родительский по отношению к вставляемому, а второй аргумент должен быть дочерним по отношению к узлу, через который вызван метод. Если через второй аргумент передать null, метод insertBefore() сработает так же, как и appendChild(), вставив узел в конец списка.

Ниже приведена простая функция, вставляющая узел на основе числовой индексации. В данной функции демонстрируется использование методов appendChild() и insertBefore().

```
// Вставка дочернего узла в позицию n
function insertAt(parent, child, n) {
  if (n < 0 || n > parent.childNodes.length)
    throw new Error("Неправильный индекс");
  else if (n == parent.childNodes.length)
    parent.appendChild(child);
  else
    parent.insertBefore(child,parent.childNodes[n]);
}
```

Если вызвать метод appendChild() или insertBefore() для вставки узла, который уже существует в документе,

этот узел будет автоматически удален из текущей позиции и вставлен в новую позицию; нет необходимости удалять его явно.

Метод `removeChild()` удаляет узел из дерева документа. Но будьте осторожны: вызывать этот метод нужно не через удаляемый узел, а через родительский. В качестве аргумента методу нужно передать дочерний удаляемый узел. Например, чтобы удалить из документа узел `n`, выполните следующую инструкцию.

```
n.parentNode.removeChild(n);
```

Метод `replaceChild()` удаляет один дочерний узел и вставляет на его место новый. Этот метод нужно вызвать через родительский узел, передав ему новый узел в качестве первого аргумента и замещаемый узел — в качестве второго. Например, чтобы заменить узел `n` строкой текста, нужно выполнить следующий код.

```
var t = document.createTextNode("[ REDACTED ]");
n.parentNode.replaceChild(t, n);
```

В приведенной ниже функции демонстрируется использование метода `replaceChild()`.

```
// Замещение узла n элементом <b> и вставка n
// в качестве дочернего узла
function embolden(n) {
  // Если n является строкой, интерпретировать
  // ее как идентификатор элемента
  if (typeof n == "string")
    n = document.getElementById(n);
  // Создание элемента <b>
  var b = document.createElement("b");
  // Замена n элементом <b>
  n.parentNode.replaceChild(b, n);
  // Вставка n как потомка <b>
  b.appendChild(n);
}
```

Стили элементов

Таблицы CSS используются для задания визуального представления HTML-документов. Веб-дизайнеры применяют их для точной настройки шрифтов, цветов, полей, отступов, рамок и позиции элемента в документе. Но таблицы CSS интересны также для программистов, создающих код JavaScript на стороне клиента, потому что стилями CSS можно манипулировать в сценарии. В данном разделе рассматривается управление стилями CSS посредством сценария. Предполагается, что вы немного знакомы с таблицами CSS.

Наиболее прямолинейный способ манипулирования стилями CSS в сценарии состоит в изменении атрибута `style` конкретного элемента. Как и большинство атрибутов, атрибут `style` является свойством объекта `Element`. Этим свойством можно манипулировать в коде JavaScript. Однако свойство `style` немного необычное: его значение — не строка или какое-либо другое примитивное значение, а объект типа `CSSStyleDeclaration`. В коде JavaScript свойства объекта стиля представляют свойства CSS, заданные в атрибуте `style`. Например, чтобы вывести текст элемента `e` крупным полужирным синим шрифтом, нужно выполнить приведенные ниже инструкции JavaScript, присваивающие заданные значения свойствам `font-size`, `font-weight` и `color`.

```
e.style.fontSize = "24pt";
e.style.fontWeight = "bold";
e.style.color = "blue";
```

В именах многих свойств стилей CSS, таких как `font-size`, содержатся дефисы. В коде JavaScript дефис интерпретируется как знак "минус", поэтому приведенное ниже выражение неправильное.

```
e.style.font-size = "24pt"; // Ошибка!
```

Следовательно, имена свойств объекта CSSStyleDeclaration должны немного отличаться от имен фактических свойств CSS. Если имя свойства CSS содержит один или несколько дефисов, имя свойства CSSStyleDeclaration создается путем удаления все дефисов и преобразования в верхний регистр каждой буквы, расположенной после дефиса. Например, свойство CSS border-left-width доступно в коде JavaScript как свойство borderLeftWidth. Кроме того, если имя свойства CSS совпадает с зарезервированным словом JavaScript (например, float), то к нему добавляется префикс css.

При работе со свойствами стилей объекта CSSStyleDeclaration не забывайте, что все значения должны задаваться в виде строк. Кроме того, все свойства, определяющие позиционирование, задаются в некоторых единицах. Следовательно, приведенные ниже инструкции, присваивающие значения свойству left, неправильные.

```
// Неправильно! Это число, а не строка
e.style.left = 300;
// Неправильно! Опущены единицы
e.style.left = "300";
```

Единицы необходимы при установке свойств стилей в JavaScript точно так же, как свойств стилей CSS. Ниже приведена правильная инструкция, присваивающая значение 300 пикселей свойству left элемента e.

```
e.style.left = "300px";
```

Если нужно присвоить свойству left вычисляемое значение, добавьте единицы измерения в конец строки путем конкатенации.

```
e.style.left = (x0 + margin + border + padding) + "px";
```

Обратите внимание на то, что числовой результат преобразуется в строковый автоматически вследствие побочного эффекта оператора +.

Атрибут style элемента HTML задает *строчный* (inline) стиль (в отличие от блочного) и переопределяет любой стиль, заданный во внешней или внедренной таблице CSS. Строчные стили обычно полезны для установки параметров отображения, как в приведенном выше примере. Программа может считывать свойства объекта CSSStyleDeclaration, представляющие строчные стили, но они возвращают осмысленные значения, только если до этого были установлены кодом JavaScript или в данном элементе HTML есть строчный атрибут style, устанавливающий указанные свойства. Например, в документе может быть определена таблица стилей, устанавливающая левое поле всех абзацев равным 30 пикселям, но если попытаться прочитать в коде свойство marginLeft одного из элементов абзаца, оно вернет пустую строку. Чтобы свойство marginLeft вернуло осмысленную строку, нужно, чтобы атрибут style был определен в конкретном элементе абзаца.

Иногда легче прочитать или записать строчный стиль элемента как одно строковое значение, а не весь объект CSSStyleDeclaration. Для этого можно применить методы getAttribute() и setAttribute() объекта Element или свойство cssText объекта CSSStyleDeclaration.

```
// Присвоение строки s атрибуту style элемента e
e.setAttribute("style", s);
e.style.cssText = s; // Другой способ

// Чтение строчного стиля элемента e
s = e.getAttribute("style");
s = e.style.cssText; // Другой способ
```

Вместо манипулирования отдельными стилями CSS посредством строчного свойства style можно управлять значением атрибута class. При изменении атрибута class элемента HTML изменяется набор селекторов стилей, применяемых к элементу. Таким способом можно изме-

нять много свойств CSS одновременно. Например, предположим, что нужно привлечь внимание пользователя к некоторому абзацу (или другому элементу) документа. Можно начать с определения яркого стиля для любого элемента класса attention.

```
.attention {/*Яркий стиль, привлекающий внимание*/
  background-color: yellow; /* Желтый цвет */
  font-weight: bold;        /* Полужирный шрифт */
  border: solid black 2px;  /* Черная рамка */
}
```

В JavaScript идентификатор class является зарезервированным словом, поэтому атрибут class элемента HTML доступен в коде JavaScript по имени className. Ниже приведен код, устанавливающий и очищающий свойство className элемента для добавления и удаления класса attention данного элемента.

```
function grabAttention(e) {
  e.className = "attention";
}
function releaseAttention(e) {
  e.className = "";
}
```

Элемент HTML может быть членом более одного класса CSS, тогда атрибут class содержит список имен классов, разделенных пробелами. Поэтому имя свойства className может ввести в заблуждение: на самом деле в нем может быть много имен классов, поэтому лучше бы оно называлось во множественном числе — classNames. В приведенных выше функциях предполагается, что свойство className определяет одно имя класса (возможно, пустое). С несколькими классами эти функции не работают. Если элементу присвоен какой-либо класс, то вызов функции grabAttention() приведет к переопределению данного класса.

В HTML5 эта проблема решается путем определения свойства `classList` для каждого элемента. Значением этого свойства служит массивоподобный (см. главу 6) объект `DOMTokenList`, элементы которого содержат имена классов элемента. Кроме элементов массива, объект `DOMTokenList` содержит методы `add()` и `remove()` для добавления и удаления отдельных имен классов из атрибута `class` элемента HTML. Метод `toggle()` работает как переключатель: он добавляет имя указанного класса, если его нет, или удаляет его, если оно существует. Метод `contains()` проверяет, содержит ли атрибут `class` указанное имя класса.

Как и другие типы коллекций DOM, объект `DOMTokenList` представляет набор классов элемента динамически, а не статически, т.е. он автоматически обновляется при изменении элемента. Если извлечь объект `DOMTokenList` из свойства `classList` элемента, а затем изменить свойство `className` этого же элемента, изменение будет немедленно отражено в списке имен классов. Аналогично этому любые изменения в списке классов немедленно видны в свойстве `className`.

Геометрия и прокрутка

До сих пор мы говорили о документе как об абстрактном дереве элементов и текстовых узлов. Но когда браузер отображает документ в окне, он создает визуальное представление документа, в котором каждый элемент имеет некоторые размеры и занимает определенную позицию. Довольно часто веб-приложение считают документ абстрактным деревом элементов и не учитывают, как эти элементы отображаются на экране. Но иногда для сценария важно учитывать точную геометрию элемента. Для задания позиций элементов часто используются таблицы CSS. Если нужно применить CSS для динамического пози-

ционирования элемента (например, для вывода окна подсказки или создания выноски) рядом с другим элементом, позиционированным браузером, то, в первую очередь, необходимо выяснить положение этого элемента.

Позиция элемента измеряется в пикселях. Координата *x* увеличивается слева направо, а *y* — сверху вниз. Существуют две разные точки, которые можно использовать в качестве начала системы координат: верхний левый угол документа или верхний левый угол клиентской области окна, в котором отображен документ (в первом случае они называются *координатами документа*, а во втором — *оконными координатами*). В окне или вкладке верхнего уровня *клиентская область* (viewport) — это часть окна браузера (исключая панель инструментов, меню, вкладки и т.п.), в которой фактически отображен документ. Если документ отображен во фрейме, то клиентской областью служит элемент `<iframe>`. В любом случае, говоря о позиции элемента, важно четко понимать, о каких координатах идет речь: документа или оконных.

Если документ меньше клиентской области окна или если документ не прокручивался, то верхний левый угол документа совпадает с верхним левым углом клиентской области. Соответственно, две указанные системы координат совпадают одна с другой. Однако в общем случае для преобразования одной системы координат в другую необходимо добавить или вычесть *смещение прокрутки*. Например, если в системе координат документа *y*=200 пикселей и пользователь прокрутил окно вниз на 75 пикселей, то элемент имеет координату *y*=125 пикселей. Аналогично, если элемент имеет оконную координату *x*=400 пикселей и пользователь прокрутил окно на 200 пикселей по горизонтали, то в системе документа координата *x* равна 600 пикселей.

Координаты документа более фундаментальные, чем оконные, потому что они не зависят от прокрутки. Тем не менее в программах на стороне клиента часто использу-

ются оконные координаты. При позиционировании элемента с помощью стилей CSS используются координаты документа. Но при запросе позиции элемента код получает оконные координаты. Аналогично при регистрации обработчика событий мыши код получает оконные координаты указателя.

Чтобы преобразовать систему координат, необходимо получить позицию ползунка прокрутки в окне браузера. Два значения позиции ползунка можно извлечь из свойств `pageXOffset` и `pageYOffset` объекта `Window`.

Иногда полезно выяснить размеры клиентской области окна, например, чтобы вычислить, какая часть документа видна в данный момент. Эти размеры можно извлечь из свойств `innerWidth` и `innerHeight` объекта `Window`.

Для выяснения размеров и позиции элемента вызовите его метод `getBoundingClientRect()`. Никаких аргументов он не ожидает, а возвращает объект со свойствами `left` (слева), `right` (справа), `top` (сверху) и `bottom` (снизу). Свойства `left` и `top` содержат координаты *x* и *y* левого верхнего угла элемента, а свойства `right` и `bottom` — соответствующие координаты правого нижнего угла.

Метод `getBoundingClientRect()` возвращает позиции в оконной системе координат. Чтобы преобразовать их в координаты документа (которые остаются неизменными даже после прокрутки окна пользователем), добавьте смещение прокрутки.

```
// Получение оконных координат
var box = e.getBoundingClientRect();
// Преобразование в координаты документа
var x = box.left + window.pageXOffset;
var y = box.top + window.pageYOffset;
```

Когда элемент отображается браузером, содержимое элемента окружено невидимыми пустыми областями, которые называются *отступами* (paddings). Отступ окружен

рамкой (не обязательно), а рамка окружена невидимыми *полями* (margins). Координаты, возвращаемые методом `getBoundingClientRect()`, включают рамку и отступы элемента, но не включают поля.

Выше мы видели, что получить позицию ползунков прокрутки можно с помощью свойств `pageXOffset` и `pageYOffset`. Если нужно установить позицию ползунков, вызовите метод `scrollTo()` через объект `Window`. Этот метод принимает значения *x* и *y* в координатах документа и применяет их в качестве смещения прокрутки, т.е. прокручивает окно таким образом, что указанные координаты совпадают с левым верхним углом клиентской области окна. Если задать точку, слишком близкую к нижнему или правому краю документа, браузер переместит документ как можно дальше, но не за пределы клиентской области.

Метод `scrollBy()` объекта `Window` напоминает метод `scrollTo()` за исключением того, что принимает относительные аргументы, добавляемые к текущему смещению ползунков.

Часто документ нужно прокрутить не на заданное расстояние, а так, чтобы некоторый элемент был виден. Для этого можно извлечь позицию элемента с помощью метода `getBoundingClientRect()`, преобразовать ее в координаты документа и вызвать метод `scrollTo()`. Но проще вызвать метод `scrollIntoView()` через элемент HTML, который нужно показать на экране. По умолчанию метод `scrollIntoView()` пытается поместить верхний край элемента как можно ближе к верхнему краю клиентской области экрана. Если передать методу единственный аргумент `false`, он попытается сделать то же самое с нижним краем элемента, поместив его как можно ближе к нижнему краю клиентской области. Одновременно браузер прокрутит клиентскую область по горизонтали таким образом, чтобы сделать элемент видимым.

Обработка событий

В приложениях JavaScript на стороне клиента используется асинхронная программная модель, управляемая событиями. В этой модели браузер генерирует событие каждый раз, когда происходит что-либо важное с документом, браузером, объектом или элементом. Например, событие генерируется, когда завершается загрузка документа, когда пользователь наводит указатель на гиперссылку, когда пользователь нажимает клавишу и т.п. Если приложение JavaScript заинтересовано в событиях некоторого типа, оно должно зарегистрировать одну или несколько функций, автоматически вызываемых при наступлении события данного типа.

Тип события — это строка, идентифицирующая событие. Например, тип `mousemove` (движение мыши) означает, что пользователь переместил мышь; тип `keydown` (клавиша вниз) означает, что пользователь нажал клавишу на клавиатуре; а тип `load` (загрузка) — что закончилась загрузка документа или другого сетевого ресурса. Поскольку тип идентифицирует событие, его часто называют *именем события*.

Целевой узел события — это узел (элемент или объект) документа, в котором произошло событие или который связан каким-либо образом с событием. Говоря о событии, всегда нужно указывать его имя и целевой узел. Например, событие `load` происходит в объекте `Window`, а событие `click` — в элементе `<button>`. На стороне клиента события чаще всего происходят в объектах `Window`,

Document и Element, но некоторые события происходят в объектах других типов. Например, в главе 13 рассматривается событие readystatechange, генерируемое объектом XMLHttpRequest.

Обработчик события — это функция, реагирующая на событие. Приложение регистрирует обработчик в браузере, задавая тип и целевой объект события. При возникновении события заданного типа в заданном целевом объекте браузер автоматически вызывает обработчик.

Объект события — это объект, создаваемый браузером, ассоциированный с событием и содержащий подробную информацию о нем. Объект события передается через аргумент функции-обработчика события. Все объекты событий имеют свойство type, задающее тип события, и свойство target, задающее целевой узел события. В каждом типе событий определен набор свойств ассоциированного объекта события. Например, объект события мыши содержит координаты указателя, а объект события клавиатуры содержит подробную информацию о нажатой клавише и о состоянии в данный момент клавиш-модификаторов. Для некоторых типов событий определено лишь несколько стандартных свойств, таких как type и target. Для этих событий важен сам факт их возникновения, а не информация о них.

Распространение события — это процесс генерации цепочки событий браузером в связанных объектах. События, специфичные для единственного объекта (например, окончание загрузки страницы — событие load в объекте Window), не распространяются. Некоторые события, возникшие в элементе документа, распространяются вверх по дереву документа (этот процесс называется *всплыванием* события). Например, когда пользователь наводит указатель на гиперссылку, событие mousemove сначала возникает

в элементе <a>, определяющем гиперссылку. После этого событие генерируется в контейнере, например в элементе <p>, затем — в элементе и наконец — в объекте Document. Иногда удобнее зарегистрировать один обработчик события в объекте Document или другом контейнере, чем непосредственно в элементе, в котором оно возникло. Обработчик может остановить распространение события, в результате чего оно не будет запускать обработчики в вышестоящих контейнерах.

Обработчик можно зарегистрировать в контейнере таким образом, чтобы он перехватил событие до того, как оно будет сгенерировано в целевом узле (подробнее об этом — далее).

С некоторыми событиями ассоциированы определенные действия, установленные для них по умолчанию. Например, после щелчка на гиперссылке таким действием является загрузка новой страницы браузером. Обработчик события может предотвратить это действие, вызвав метод объекта события.

Типы событий

В данном разделе рассматривается несколько категорий событий: события формы, мыши, клавиатуры и др. Каждая категория содержит ряд типов событий. Далее в каждом подразделе описаны события определенной категории, а также рассмотрены важные свойства объектов событий данной категории

События формы

Формы и гиперссылки исторически были первыми элементами, которыми манипулировали сценарии JavaScript на заре Интернета. Благодаря этому события формы наиболее стабильные и поддерживаются лучше

событий других категорий. Элементы <form> генерируют события передачи и сброса данных формы. Кнопки, переключатели и флажки генерируют события щелчков на них. Элементы формы, поддерживающие состояния, в общем случае генерируют события изменения состояний, когда пользователь, например, вводит текст, выбирает пункт или отмечает флажок. В текстовых полях событие изменения состояния не генерируется до тех пор, пока пользователь не заканчивает работать с ними и не передает фокус другому элементу. Элементы формы реагируют на получение и потерю фокуса двумя отдельными событиями.

С событиями передачи и сброса формы ассоциированы выполняемые по умолчанию действия, которые можно отменить в обработчике события. Аналогичные действия ассоциированы с событиями щелчков. События получения и потери фокуса (в отличие от всех других событий формы) не всплывают.

События окна

События окна ассоциированы непосредственно с окном браузера, а не с отображаемым содержимым документа (важно отметить, что имена некоторых из этих событий совпадают с именами событий документа).

Наиболее важное событие окна — `load` (загрузка). Оно генерируется, когда документ и все его внешние ресурсы (такие, как изображения) полностью загружены и содержимое документа отображено на экране. Альтернативные события — `DOMContentLoaded` и `readystatechange`. Они возникают немного раньше события `load` — когда документ и его элементы доступны для манипулирования пользователем, но перед загрузкой внешних ресурсов.

Событие `unload` противоположно событию `load`: оно генерируется, когда пользователь переходит к другому документу. Обработчик события `unload` можно использовать для сохранения пользовательских состояний, но с его помощью нельзя отменить переход к другому документу. Событие `beforeunload` похоже на `unload`, но предоставляет возможность попросить пользователя подтвердить, действительно ли он хочет перейти на другую веб-страницу. Если обработчик `beforeunload` возвращает строку, она будет отображена для пользователя в диалоговом окне перед загрузкой новой страницы, и пользователь получит возможность отменить переход и оставить на экране прежнюю страницу.

События получения и потери фокуса элементами формы также используются как события окна. Они генерируются, когда текущее окно браузера получает или теряет фокус ввода с клавиатуры.

И наконец, события изменения размеров и прокручивания генерируются, когда пользователь выполняет эти операции с помощью мыши. Событие прокручивания может возникнуть также в любом прокручиваемом элементе документа, в котором установлено свойство `overflow` стиля CSS.

События мыши

Эти события генерируются, когда пользователь перемещает мышь или щелкает в окне документа. События мыши генерируются даже для наиболее глубоко вложенных элементов, на которые наведен указатель, но в этом случае они всплывают вверх по дереву документа. Объект события мыши, передаваемый обработчику, содержит набор свойств, описывающих позицию указателя, состояние кнопок мыши и состояние клавиш-модификаторов

в момент возникновения события. Свойства clientX и clientY определяют позицию указателя в оконной системе координат. Свойства altKey, ctrlKey, metaKey и shiftKey равны true, если в момент возникновения события соответствующая клавиша-модификатор была нажата. Свойство detail сообщает о типе щелчка: одиночный, двойной или тройной.

Событие mousemove генерируется с большой частотой, когда пользователь перемещает мышь или перетаскивает что-либо. Поскольку это событие генерируется часто, в его обработчик нельзя закладывать трудоемкие задачи. События mousedown и mouseup возникают, когда пользователь нажимает и отпускает кнопку мыши. Зарегистрировав обработчик mousedown, который регистрирует обработчик mousemove, можно обнаружить перетаскивание и отреагировать на него. Но для этого нужно, чтобы программа продолжала перехватывать события mousemove, даже когда указатель выйдет за пределы элемента, в котором началось перетаскивание.

После событий mousedown и mouseup браузер генерирует событие click. Если пользователь щелкает на кнопке мыши дважды (через достаточно короткий промежуток времени), то после второго события click браузер генерирует событие двойного щелчка dblclick. После щелчка правой кнопкой мыши браузер часто отображает контекстное меню. Обычно событие contextmenu генерируется перед появлением контекстного меню на экране, поэтому в обработчике данного события можно отменить его отображение. Это же событие можно использовать для извещения о щелчке правой кнопкой мыши.

Когда пользователь перемещает указатель, в момент вхождения указателя в область элемента браузер генерирует событие mouseover данного элемента. Когда указатель

выходит за пределы элемента, браузер генерирует событие mouseout данного элемента. Объекты этих событий имеют свойство relatedTarget, указывающее на другой элемент, вовлеченный в перемещение. События mouseover и mouseout всплывают вверх по дереву документа, подобно всем события мыши. Часто это неудобно, потому что при запуске обработчика mouseout нужно выяснить, действительно ли указатель покинул интересующий нас элемент или он всего лишь переместился с одного дочернего элемента на другой. В этом случае лучше использовать события mouseenter и mouseleave — новые невсплывающие версии событий mouseover и mouseout, поддерживаемые последними браузерами.

Когда пользователь прокручивает колесико мыши, браузер генерирует событие mousewheel. Объект события, передаваемый обработчику, содержит свойства, сообщающие об угле и направлении поворота.

События клавиатуры

Имея фокус, браузер генерирует события клавиатуры каждый раз, когда пользователь нажимает или отпускает клавишу. Однако важно учитывать, что горячие клавиши операционной системы или браузера часто перехватываются ими и не доходят до интерпретатора JavaScript. События клавиатуры генерируются в любом элементе документа, имеющем фокус клавиатуры, и всплывают по дереву вверх до окна или документа. Если ни один элемент не имеет фокуса клавиатуры, событие генерируется непосредственно в документе. Обработчики событий клавиатуры получают объект события с полем keyCode, сообщающим, какая клавиша была нажата или отпущена. Кроме поля keyCode, в объекте события клавиатуры есть также свойства altKey, ctrlKey, metaKey и shiftKey, отражающие

состояние клавиш-модификаторов в момент возникновения события.

События keydown и keyup являются низкоуровневыми и генерируются, когда пользователь нажимает или отпускает любую клавишу, включая клавиши-модификаторы. Когда событие keydown генерирует печатный символ, дополнительно возникает событие keypress (после keydown, но до keyup). Если пользователь нажал и удерживает клавишу, событие keypress повторяется многократно, пока пользователь не отпустит клавишу. Событие keypress высокоуровневое и текстовое. В его объекте события установлен сгенерированный символ, а не нажатая клавиша. В некоторых браузерах (в частности, в Firefox) необходимо использовать свойство charCode объекта события keypress вместо свойства keyCode.

События keydown, keyup и keypress поддерживаются всеми браузерами, но все же их использование порождает некоторые проблемы, связанные с тем, что значения свойства keyCode не очень хорошо стандартизированы.

События HTML5

Спецификация HTML5 определяет ряд новых программных интерфейсов для веб-приложений. Во многих из них определены события. В данном разделе приведены список и краткое описание событий, введенных в HTML5. Одни из этих событий можно использовать уже сейчас, а другие реализованы в браузерах еще не полностью.

Одно из наиболее разрекламированных новых средств HTML5 — элементы <audio> и <video>, предназначенные для воспроизведения звука и видео. С этими элементами ассоциирован длинный список событий, служащих для передачи сообщений о сетевых событиях, статусе буфера данных, статусе воспроизведения и т.д. Ниже приведен список этих событий.

```
canplay
canplaythrough
durationchange
emptied
ended
loadeddata
loadedmetadata
loadstart
pause
play
playing
progress
ratechange
seeked
seeking
stalled
suspend
timeupdate
volumechange
waiting
```

Этим медиасобытиям передаются обычные объекты событий без специальных свойств. Свойство `target` идентифицирует элемент `<audio>` или `<video>`, который имеет особые свойства и методы.

Спецификация HTML5 API, связанная с перетаскиванием, позволяют приложениям JavaScript принимать участие в управляемых операционной системой операциях перетаскивания и обмена данными между веб-приложениями и настольными программами. В спецификации определены следующие семь событий перетаскивания.

```
dragstart
drag
dragend
dragenter
dragover
dragleave
drop
```

События перетаскивания генерируются с объектами событий, аналогичными событиям мыши. Дополнительное свойство `dataTransfer` содержит объект `DataTransfer`, предоставляющий информацию о передаваемых данных и доступных форматах.

Спецификация HTML5 определяет механизм управления историей, позволяющий веб-приложению взаимодействовать с кнопками **Вперед** и **Назад** браузера. В механизме управления историей используются события `hashchange` и `popstate`, служащие для извещения об этапах жизненного цикла документа, таких как загрузка или выгрузка. Эти события генерируются объектом `Window`, а не отдельными элементами документа.

В HTML5 определено много новых средств форм. Кроме стандартизации событий ввода, в HTML5 добавлен механизм проверки формы с помощью события ошибки, генерируемого в элементе формы, не прошедшем проверку.

Кроме того, в HTML5 включена поддержка автономных веб-приложений, которые можно устанавливать локально в кеше приложений и выполнять, когда браузер отключен от сети (например, когда мобильное устройство находится вне зоны покрытия сети). Два наиболее важных события этого типа — `offline` и `online`. Они возникают в объекте `Window`, когда браузер устанавливает или разрывает сетевое соединение. Ряд дополнительных событий определены для извещения о прогрессе загрузки приложения и обновлениях кеша.

```
cached
checking
downloading
error
noupdate
obsolete
```

```
progress
updateready
```

В некоторых новых библиотеках веб-приложений для асинхронной коммуникации между приложениями используется событие `message`. Спецификация Cross-Document Messaging API позволяет сценариям, определенным в документах на разных серверах, обмениваться сообщениями меду собой. Это позволяет обойти правило ограничения домена (см. главу 10) безопасным способом. Каждое переданное сообщение приводит к возникновению события `message` в окне принимающего документа. Передаваемый обработчику объект события имеет свойство `data`, в котором находится содержимое сообщения, и свойства `source` и `origin`, идентифицирующие передатчик сообщения. Аналогичным образом событие `message` используется для взаимодействия с потоками Web Workers и для обмена данными посредством протоколов Server-Sent Events и WebSockets.

В HTML5 и других родственных стандартах определен ряд событий, возникающих в объектах, отличных от окон, документов и элементов. В версии 2 спецификации `XMLHttpRequest` и спецификации File API определены события, отслеживающие асинхронные операции ввода-вывода. Эти события генерируются в объектах `XMLHttpRequest` и `FileReader`. Каждая операция чтения начинается с возникновения события `loadstart`, после которого генерируются события прогресса и событие `loadend`. Кроме того, каждая операция заканчивается событием `load`, `error` или `abort`, возникающим непосредственно перед событием `loadend`.

И наконец, в HTML5 и родственных стандартах определен ряд дополнительных категорий событий. В спецификации Web Storage API определены события объекта

Window, извещающие об изменении хранящихся данных. Стандартизированы также события beforeprint и afterprint, которые сначала появились в браузерах Internet Explorer. Эти события возникают в объекте Window непосредственно до и после печати документа и позволяют добавить или удалить содержимое, например дату и время печати документа. Эти события не рекомендуется использовать для управления представлением печатной версии документа, потому что для этой цели существуют средства CSS.

События сенсорных экранов и мобильных устройств

Широкое распространение мобильных устройств, оснащенных сенсорными экранами, потребовало создания новых категорий событий. Во многих случаях сенсорные события связываются с традиционными типами событий, такими как щелчки или прокручивание. Но не каждое действие на сенсорном экране эмулирует мышь, и, наоборот, не каждое касание можно интерпретировать как событие мыши. В данном разделе кратко рассматриваются события жестов и касаний, генерируемые браузером Safari на устройствах iPhone и iPad компании Apple. Кроме того, рассматривается событие изменения ориентации, возникающее, когда пользователь поворачивает устройство.

Браузер Safari генерирует события масштабирования и поворота для жестов двумя пальцами. Событие gesturestart генерируется в момент начала жеста, а событие gestureend — в конце. Между этими двумя событиями браузер генерирует последовательность событий gesturechange, отслеживающих жест. Объекты этих событий содержат числовые свойства scale и rotation. Значение scale равно отношению текущего расстояния между пальцами к начальному расстоянию. Для жеста сворачивания значение scale меньше 1, а для жеста разворачивания —

больше 1. Значение `rotation` равно углу поворота двух пальцев с начала события в градусах. Положительное значение означает поворот по часовой стрелке.

События жестов — это высокоуровневые события, извещающие программу об операциях, интерпретированных браузером и операционной системой. Если нужно создать собственные жесты, используйте низкоуровневые события касания. Когда палец прикасается к экрану, возникает событие `touchstart`. Когда палец двигается, операционная система с высокой частотой генерирует события `touchmove`. Когда палец поднимается, генерируется событие `touchend`. В отличие от событий мыши события касания не передают непосредственно координаты точки прикосновения. Вместо этого объект события касания имеет свойство `changedTouches`, содержащее массивоподобный объект, элементы которого описывают позицию касания.

Событие `orientationchanged` генерируется в объекте `Window` устройством и позволяет переключать ориентацию экрана с портретной на альбомную и обратно. Объект события `orientationchanged` не очень полезный. Например, в мобильной версии Safari текущая ориентация находится в свойстве `orientation` объекта `Window` как одно из четырех чисел: `0`, `90`, `180` или `-90`.

Регистрация обработчика события

Существуют два способа регистрации обработчика. Первый — установка свойства целевого объекта или документа. Второй — передача обработчика методу целевого объекта или элемента. Чтобы задача была еще сложнее, существует также по две версии каждого способа. Свойство обработчика события можно установить в коде JavaScript, или (для элемента) можно установить соответствующий атрибут непосредственно в коде HTML.

Установка свойства обработчика

Простейший способ регистрации обработчика состоит в присвоении функции-обработчика свойству целевого объекта события. Соглашения об именовании требуют, чтобы имя свойства обработчика начиналось с префикса on, после которого должно находиться имя события: onclick, onchange, onload, onmouseover или т.п. Обратите внимание на то, что имена свойств чувствительны к регистру и должны быть приведены в нижнем регистре, даже когда имя события состоит из многих слов (например, readystatechange). Ниже представлен пример регистрации обработчиков двух событий.

```
// Установка свойства onload объекта Window.
// Функция является обработчиком и вызывается
// при загрузке документа.
window.onload = function() {
  // Извлечение элемента <form>.
  var elt = document.getElementById("address");
  // Регистрация функции-обработчика, которая будет
  // вызываться до передачи формы.
  elt.onsubmit = function() { return validate(this); }
}
```

Недостаток свойства обработчика состоит в том, что в нем используется предположение, будто целевые объекты будут иметь не более одного обработчика для каждого типа события. При создании кода библиотеки, которая будет использоваться с произвольными документами, данный способ неприменим.

Установка атрибута обработчика

Свойство обработчика для элемента можно также установить как атрибут соответствующего дескриптора HTML. Значением атрибута должна быть строка кода JavaScript, содержащая только тело функции обработчика

без полного объявления функции. Это означает, что код обработчика не должен быть заключен в фигурные скобки и перед ним не должно быть ключевого слова function, как показано ниже.

```
<button onclick="alert('Привет!');">
  Щелкните здесь
</button>
```

Если атрибут обработчика содержит много инструкций JavaScript, они должны быть разделены точками с запятыми. Можно переносить длинный код обработчика в другие строки документа.

Некоторые типы событий направляются браузеру, а не отдельному элементу документа. В коде JavaScript эти события нужно регистрировать в объекте Window. В HTML-коде их нужно помещать в дескриптор <body>, а браузер зарегистрирует их в объекте Window. Ниже приведен полный список таких событий, определенных в проекте спецификации HTML5.

```
onafterprint
onbeforeprint
onbeforeunload
onblur
onerror
onfocus
onhashchange
onload
onmessage
onoffline
ononline
onpagehide
onpageshow
onpopstate
onredo
onresize
onstorage
onundo
onunload
```

При задании строки кода JavaScript как значения атрибута обработчика браузер преобразует ее в функцию, которая выглядит приблизительно так.

```
function(event) {
  with(document) {
    with(this.form || {}) {
      with(this) {
        /* Здесь находится код функции */
      }
    }
  }
}
```

Инструкция `with` и аргумент `event` подробнее рассматриваются при обсуждении вызова обработчика.

Метод addEventListener()

В любом объекте, который может быть целевым для события (включая `Window`, `Document` и все узлы `Element`), определен метод `addEventListener()`. Его можно использовать для регистрации обработчика события данного целевого объекта. Метод `addEventListener()` принимает три аргумента. Первый — тип события, для которого регистрируется обработчик. Тип (или имя) — это строка, в которой не должно быть префикса `on`, используемого при установке свойства обработчика. Второй аргумент — функция, вызываемая при возникновении события данного типа (т.е. обработчик). Третий аргумент — необязательное булево значение. По умолчанию применяется значение `false`. Если передать значение `true`, функция будет зарегистрирована как *перехватывающий* обработчик, вызываемый на другом этапе диспетчеризации события (подробнее об этом — далее).

Приведенный ниже код регистрирует два обработчика для события щелчка на элементе <button>. Обратите внимание на различия между двумя способами регистрации.

```
<button id="mybutton">Щелкните здесь</button>
<script>
var b = document.getElementById("mybutton");
b.onclick = function() { alert("Привет!"); };
b.addEventListener("click",
   function() { alert("Еще раз привет!"); });
</script>
```

Вызов метода addEventListener() с первым аргументом click не влияет на значение свойства onclick. Поэтому после щелчка на кнопке приведенный выше код выведет диалоговое окно alert() два раза. Но еще важнее то, что метод addEventListener() можно вызывать много раз для регистрации нескольких обработчиков одного и того же события того же объекта. При возникновении события все зарегистрированные обработчики будут вызваны в той последовательности, в которой они были зарегистрированы. Вызов метода addEventListener() более одного раза через один и тот же объект и с одинаковыми аргументами приведет к регистрации обработчика только один раз, причем последующие регистрации этого обработчика не повлияют на его место в ряду других обработчиков.

Метод addEventListener() "идет в паре" с методом removeEventListener(), который ожидает те же аргументы, но не регистрирует, а отменяет регистрацию обработчика. Пара этих методов часто полезна для временной регистрации обработчика. Например, иногда полезно временно зарегистрировать перехватывающие обработчики событий mousemove и mouseup, чтобы увидеть, когда пользователь начнет перетаскивание. Когда перетаскивание закончится, регистрацию обработчиков можно будет отменить. Ниже приведены инструкции, отменяющие регистрацию двух обработчиков.

```
document.removeEventListener("mousemove",
                             handleMove, true);
document.removeEventListener("mouseup",
                              handleUp, true);
```

Вызов обработчика события

После регистрации обработчика браузер будет автоматически вызывать его при возникновении события указанного типа в указанном объекте. В данном разделе подробно рассматривается выполнение функции обработчика, его аргумент, контекст вызова (значение `this`), область видимости вызова и возвращаемое значение обработчика.

Кроме того, в данном разделе рассматривается *распространение* события — процесс запуска обработчика в исходном целевом объекте события и возникновение события этого же типа в узлах, расположенных выше по дереву документа.

Аргумент обработчика

Обработчики запускаются с единственным аргументом, содержащим объект события. Свойства объекта события (описанные выше) предоставляют обработчику всю нужную информацию о событии.

При регистрации события путем установки атрибута в целевом элементе HTML (см. выше) браузер преобразует строку кода JavaScript в функцию с одним аргументом, называющимся `event`. Это означает, что обработчик может ссылаться на объект события по имени `event`.

Контекст обработчика

Регистрируя обработчик путем установки свойства, вы определяете новый метод объекта.

```
e.onclick = function() { /* Код обработчика */ };
```

Следовательно, обработчик вызывается как метод объекта, в котором он определен, и в теле обработчика ключевое слово this ссылается на целевой объект события.

Обработчики, зарегистрированные с помощью метода addEventListener(), также вызываются с целевым объектом, указанным с помощью значения this.

Область видимости обработчика

Как и все функции JavaScript, обработчики событий имеют лексическую область видимости. Это означает, что обработчик выполняется в области видимости, в которой он был определен, а не в той, в которой он был вызван. Обработчику доступны все локальные переменные в его области видимости.

Однако область видимости обработчика, зарегистрированного как атрибут HTML, подчиняется другим правилам. Обработчик преобразуется в высокоуровневую функцию, которая имеет доступ только к глобальным, но не к локальным переменным. По историческим причинам обработчики выполняются в модифицированной цепочке областей видимости. Код обработчика, определенного как атрибут HTML, может использовать свойства целевого объекта, родительского объекта <form> и объекта Document таким образом, как будто эти свойства являются локальными переменными. Если обработчик события определен в атрибуте HTML, инструкция with определяет модифицированную цепочку областей видимости.

Атрибуты HTML — не очень подходящее место для длинных строк кода, поэтому модифицированная цепочка областей видимости допускает полезные сокращения. Вместо this.tagName можно писать tagName, а вместо document.getElementById — getElementById. В элементах документа, находящихся в контейнере <form>, можно ссылаться на другие

элементы формы по значениям атрибута id. Например, вместо this.form.zipcode можно написать zipcode.

С другой стороны, модифицированная цепочка областей видимости обработчика — постоянный источник коварных ошибок, потому что свойства объектов цепочки переопределяют одноименные свойства глобального объекта. Особенно эта проблема актуальна в формах, поскольку имена и значения id элементов формы определяют свойства контейнера. Например, если форма содержит элемент с атрибутом id, равным location, то во всех обработчиках формы для ссылки на объект Location текущего окна нужно вместо location писать window.location.

Возвращаемое значение обработчика

Если обработчик зарегистрирован путем установки свойства объекта или атрибута HTML, его возвращаемое значение часто играет важную роль. В общем случае возвращаемое значение false сообщает браузеру о том, что он не должен выполнять действие, установленное для события по умолчанию. Например, обработчик события onclick кнопки Submit может вернуть false, чтобы предотвратить передачу данных формы. Это весьма полезно, если процедура проверки текста, введенного пользователем, нашла ошибку. Аналогично обработчик события onkeypress поля ввода может фильтровать введенные данные, если пользователь вводит неправильные символы.

Возвращаемое значение обработчика onbeforeunload объекта Window также очень важное. Данное событие генерируется, когда пользователь переходит на новую страницу. Если обработчик возвращает строку, ее можно отобразить в модальном диалоговом окне, которое просит пользователя подтвердить его намерение закрыть старую страницу.

Важно понимать, что возвращаемые значения важны только для обработчиков, зарегистрированных как свойства. Далее будет показано, что обработчики, зарегистрированные с помощью метода `addEventListener()`, для предотвращения запуска процедуры, установленной по умолчанию, должны использовать метод `preventDefault()` объекта события.

Распространение событий

Когда целевым объектом события является окно `Window` или отдельный объект (например, `XMLHttpRequest`), браузер реагирует на событие, запустив соответствующий обработчик через этот объект. Если же объект события — документ или элемент документа, ситуация существенно усложняется.

После запуска обработчика, зарегистрированного в целевом элементе, большинство событий всплывают по дереву DOM. Сначала запускаются обработчики, зарегистрированные в родительском элементе. Затем — обработчики в следующем родительском элементе. Этот процесс продолжается, пока не дойдет до объекта `Document` и далее — до объекта `Window`. Всплывание событий создает альтернативу: вместо регистрации обработчиков во многих отдельных элементах можно зарегистрировать обработчик в одном элементе-протомке и в нем обработать событие. Например, можно зарегистрировать событие изменения в одном элементе `<form>`, а не в каждом элементе формы, в котором может произойти изменение.

Всплывают почти все события, происходящие в элементах документа. Наиболее важные исключения из этого правила — события получения фокуса, потери фокуса и прокрутки. Событие загрузки элемента всплывает, но распространение прекращается по достижении объек-

та Document, и в объект Window оно не распространяется. Событие load объекта Window генерируется только при загрузке всего документа.

Всплывание события — третья "фаза" его распространения. Вторая фаза — вызов обработчиков целевого объекта. Первая фаза, происходящая перед вызовом обработчиков целевого объекта, называется *перехватом*. Как вы помните, метод addEventListener() принимает через необязательный третий аргумент булево значение. Если оно равно true, обработчик регистрируется как перехватывающий для вызова в первой фазе распространения.

Фаза перехвата похожа на всплывание наоборот. Сначала вызываются перехватывающие обработчики объекта Window, затем — перехватывающие обработчики объекта Document, и так далее вниз по дереву DOM, пока не будет достигнут перехватывающий обработчик объекта, родительского по отношению к целевому. Перехватывающие обработчики, зарегистрированные в самом целевом объекте, не запускаются.

Процедура перехвата предоставляет возможность "подсмотреть" событие до его передачи целевому объекту. Перехватывающие обработчики обычно используются для отладки или в алгоритмах отмены событий, обеспечивающих фильтрацию для обработчиков целевого объекта, которые могут быть не запущены. Одно из популярных применений данной процедуры — обработка событий перетаскивания, когда событие движения мыши нужно обработать в перетаскиваемом объекте, а не в элементе, над которым проходит перетаскиваемый элемент.

Отмена события

Выше было показано, что возвращаемое значение обработчика, зарегистрированного как свойство, мож-

но использовать для отмены действия, установленного по умолчанию для события данного типа. С этой же целью можно вызвать метод `preventDefault()` объекта события.

Отмена действия, установленного по умолчанию, — лишь один из видов отмены событий. Можно также отменить распространение события. У объектов событий есть метод `stopPropagation()`, который можно вызвать для остановки распространения события. Если есть другие обработчики, определенные в этом же объекте, они будут вызваны, но никакие обработчики в других объектах вызваны уже не будут. Метод `stopPropagation()` можно запустить в любой момент распространения события. Он работает на фазе перехвата, в целевом объекте события и на фазе всплывания. Еще один метод объекта события — `stopImmediatePropagation()` — предотвращает распространение события в любой объект и вызов любого обработчика, зарегистрированного в этом объекте.

Сетевое взаимодействие

В данной главе рассматриваются четыре сетевые технологии на основе клиентских JavaScript-сценариев. Первая из них реализована в объекте `XMLHttpRequest`, который широко применяется в архитектуре Ajax. Это наиболее важная из четырех технологий, и ей посвящена основная часть главы. Кроме того, здесь описывается методика JSONP для сетевого взаимодействия в стиле Ajax с дескрипторами `<script>`, модель Comet с новой спецификацией EventSource API и новый полнодуплексный протокол WebSockets.

Класс XMLHttpRequest

Используемые браузерами средства HTTP определены в классе `XMLHttpRequest`. Каждый экземпляр данного класса представляет пару "запрос–ответ" в протоколе HTTP. Свойства и методы объекта `XMLHttpRequest` позволяют задать параметры запроса и прочитать данные ответа. Класс `XMLHttpRequest` часто сокращенно называют XHR. В данной главе термин "XHR 2" обозначает новую версию спецификации. Следует отметить, что имя `XMLHttpRequest` не имеет ничего общего с XML. Это просто совпадение, исторически обусловленное тем, что когда-то планировалось тесно связать данную технологию с моделью XML.

Первый шаг в использовании XHR API — создание объекта `XMLHttpRequest`.

```
var request = new XMLHttpRequest();
```

Можно также повторно использовать существующий объект `XMLHttpRequest`, но тогда будет прерван любой запрос, выполняющийся через этот объект.

Каждый HTTP-запрос состоит из четырех частей:

- метод HTTP;
- запрашиваемый URL-адрес;
- необязательный набор заголовков запроса, который может содержать, например, аутентификационную информацию;
- необязательное тело запроса.

Передаваемый сервером HTTP-ответ состоит из трех частей:

- числовой или текстовый код статуса, сообщающий об успехе или неудаче запроса;
- набор заголовков ответа;
- тело ответа.

В следующих разделах демонстрируется установка каждой из указанных частей HTTP-запроса, а также извлечение указанных частей HTTP-ответа средствами XHR API.

Базовая архитектура запросов и ответов HTTP довольно проста, и работать с ней легко. Однако на практике часто возникают разнообразные осложнения: клиент и сервер обмениваются файлами "cookie", сервер перенаправляет браузер на другие серверы, некоторые ресурсы (но не все) кешированы, некоторые клиенты передают все запросы через прокси-сервер и т.п. Класс `XMLHttpRequest` работает не на уровне протокола HTTP, а на уровне браузера. Браузер сам побеспокоится о файлах "cookie", перенаправлении, кешировании, прокси-серверах и других вещах, в результате чего код сможет работать только с запросами и ответами.

Объект XMLHttpRequest и локальные файлы

Возможность использовать относительные URL-адреса на веб-страницах означает, что разрабатывать и тестировать веб-страницы можно с помощью локальной файловой системы, а затем можно развернуть отлаженную версию на веб-сервере. Однако при использовании Ajax и `XMLHttpRequest` это чаще всего невозможно. Объект `XMLHttpRequest` предназначен для работы с протоколами HTTP и HTTPS, но не с протоколом `file://`. Это означает, что при работе с объектом `XMLHttpRequest` вам, скорее всего, придется для тестирования программы выгружать файлы на веб-сервер (в крайнем случае — на локальный).

Создание запроса

После создания объекта `XMLHttpRequest` следующий шаг в создании HTTP-запроса — вызов метода `open()` объекта `XMLHttpRequest` для задания двух обязательных частей запроса: метода и URL-адреса.

```
request.open("GET", "data.csv");
```

Первый аргумент определяет метод HTTP. Методы GET и POST поддерживаются всеми браузерами и серверами. Метод GET используется для передачи простых запросов. Этот метод предпочтителен в следующих случаях: когда строка URL может полностью определить запрашиваемый ресурс, когда запрос не создает побочных эффектов на сервере и когда ответ сервера доступен для кеширования. Метод POST позволяет передать дополнительные данные в теле запроса. Часто эти данные сохраняются на стороне сервера в базе данных.

Кроме GET и POST, спецификация `XMLHttpRequest` позволяет использовать в качестве первого аргумента метода `open()` методы DELETE, HEAD, OPTIONS и PUT.

Второй аргумент метода `open()` — строка URL, являющаяся содержимым запроса и задаваемая относительно URL-адреса документа сценария, который вызвал метод `open()`. Если задать абсолютный URL-адрес, то протокол, хост и порт адреса и документа должны совпадать. Кроссдоменные HTTP-запросы, как правило, вызывают ошибку. Однако спецификация XHR2 разрешает такие запросы, когда их явно разрешает сервер (подробнее об этом — далее).

Следующий этап — установка заголовка запроса (если это необходимо). Например, в запросе POST для задания MIME-типа тела запроса нужен заголовок `Content-Type`.

```
request.setRequestHeader("Content-Type",
                         "text/plain");
```

Если вызвать метод `setRequestHeader()` несколько раз с одним и тем же заголовком, новое значение заголовка не заменит старое. Вместо этого в HTTP-запрос будут включены копии заголовка, или заголовок будет определять много значений.

Не задавайте заголовки `Content-Length`, `Date`, `Referer` и `User-Agent`; объект `XMLHttpRequest` добавляет их автоматически и не разрешает редактировать их вручную. Кроме того, объект `XMLHttpRequest` автоматически обрабатывает файлы "cookie", задает время жизни соединения, кодовую таблицу и правила кодирования. Программисту не разрешено устанавливать эти заголовки.

Окончательный этап создания HTTP-запроса с помощью объекта `XMLHttpRequest` — задание необязательного тела запроса и его передача на сервер. Для этого нужно вызвать метод `send()`.

```
request.send(null);
```

У запросов GET нет тела, поэтому нужно передать методу `null` или опустить аргумент. У запросов POST тело

обычно есть. Его формат должен соответствовать заголовку Content-Type, заданному методом setRequestHeader().

В листинге 13.1 используется каждый из упомянутых выше методов класса XMLHttpRequest. Данный код передает методом POST строку текста на сервер и задает игнорирование любого ответа сервера. Обратите внимание на то, что строка, передаваемая в теле запроса, может быть довольно сложным объектом, например объектом JavaScript, закодированным с помощью метода JSON.stringify(), или набором пар "имя–значение", закодированных формой.

Листинг 13.1. Передача методом POST неформатированного текста на сервер

```
function postMessage(msg) {
  var r = new XMLHttpRequest(); // Создание запроса
  r.open("POST", "/log.php");   // Открытие запроса
  // Задание формата содержимого
  r.setRequestHeader("Content-Type",
                     "text/plain;charset=UTF-8");
  // Передача сообщения в теле запроса
  r.send(msg);
  // Ответ игнорируется
}
```

Обратите внимание на то, что в листинге 13.1 метод send() инициирует ответ и завершается. Следовательно, он не блокирует поток браузера ожиданием ответа сервера. Как показано в следующем разделе, HTTP-ответы являются асинхронными.

Получение ответа

Полный HTTP-ответ состоит из кода статуса, набора заголовков ответа и тела ответа. Все эти части ответа доступны посредством свойств и методов объекта XMLHttpRequest.

- Свойства `status` и `statusText` возвращают статус HTTP в числовой и текстовой формах. Эти свойства содержат стандартные значения HTTP, такие как `200` и `OK` для успешного ответа или `404` и `Not Found` (Не найден) для URL-адресов, неправильно ссылающихся на серверный ресурс.
- Заголовки ответа можно извлечь с помощью методов `getResponseHeader()` и `getAllResponseHeaders()`.
- Тело ответа доступно в текстовом формате в свойстве `responseText`.

Объект `XMLHttpRequest` работает асинхронно: метод `send()` немедленно завершается после передачи запроса, а перечисленные выше методы и свойства ответа недействительны, пока не получен ответ. Чтобы получить уведомление о поступлении ответа, необходимо включить прослушивание события `readystatechange` объекта `XMLHttpRequest` (в XHR 2 можно использовать события прогресса, описанные далее). Готовность ответа к обработке определяется значением свойства `readyState`.

Целочисленное значение свойства `readyState` определяет статус HTTP-запроса. Ниже приведены его возможные значения.

Значение	Описание
0	Метод `open()` еще не вызывался
1	Вызван метод `open()`
2	Получены заголовки ответа
3	Получено тело ответа
4	Передача ответа завершена

Чтобы включить прослушивание событий `readystatechange`, присвойте обработчик события свойству `onready-`

statechange объекта XMLHttpRequest или вызовите метод addEventListener(). В листинге 13.2 приведена функция getText(), демонстрирующая прослушивание события readystatechange. Сначала обработчик убеждается в том, что запрос завершен. Затем он проверяет код статуса ответа, убеждаясь в том, что запрос был успешным. Затем он просматривает заголовок Content-Type, чтобы проверить, правильный ли тип ответа. Если все три условия удовлетворены, обработчик передает тело ответа как текст заданной функции обратного вызова. Эта функция обработает ответ, передав его, например, методу JSON.parse().

Листинг 13.2. Получение HTTP-ответа

```
// Создание HTTP-запроса GET к заданному URL-адресу.
// При поступлении ответа код проверяет формат и
// передает текст заданной функции обратного вызова
function getText(url, callback) {
  var r = new XMLHttpRequest(); // Создание запроса
  r.open("GET", url);           // Задание URL-адреса
  r.onreadystatechange = function() {
    // Проверка статуса запроса
    if (r.readyState === 4 && r.status === 200) {
       var type = r.getResponseHeader("Content-Type");
       // Проверка формата ответа
       if (type.match(/^text/))
          callback(r.responseText);
    }
  };
  r.send(null); // Передача запроса
}
```

HTTP-события прогресса

В предыдущих примерах событие readystatechange использовалось для определения момента завершения HTTP-запроса. В проекте спецификации XHR 2 определен более полезный набор событий. В новой модели событий объект

XMLHttpRequest генерирует на разных фазах запроса разные типы событий, в результате чего проверять состояние свойства readyState больше нет необходимости.

Если браузер поддерживает спецификацию XHR 2, новые события генерируются в такой последовательности. При вызове метода send() генерируется одно событие loadstart. Когда сервер выгружает ответ, объект XMLHttpRequest многократно генерирует события progress, обычно каждые 50 мс. Эти события можно использовать для оповещения пользователя о прогрессе запроса. Если запрос завершается быстро, события progress могут не возникнуть ни разу. При завершении запроса возникает событие load.

Завершенный запрос не обязательно успешный, поэтому обработчик события load должен проверить код статуса в свойстве status объекта XMLHttpRequest, дабы убедиться в том, что получен ответ 200 OK, а не, например, 404 Not Found.

Существуют три вида неудачных HTTP-запросов и три соответствующие причины. По истечении времени запроса генерируется событие timeout. Если запрос прерван, генерируется событие abort. И наконец, ошибка в сети (например, превышение допустимого количества перенаправлений) может предотвратить завершение запроса и привести к возникновению события error.

Объект события progress имеет три дополнительных полезных свойства (кроме обычных свойств объекта Event, таких как type и timestamp). Свойство loaded содержит количество байтов, переданных на данный момент. Свойство total равно общей длине (в байтах) передаваемых данных, указанной в заголовке Content-Length. Если этого заголовка нет, свойство total равно 0. Свойство lengthComputable равно true, если длина содержимого из-

вестна, и false — в противном случае. В обработчиках событий progress особенно полезны свойства total и loaded.

```
request.onprogress = function(e) {
  if (e.lengthComputable) {
    var p = Math.round(100*e.loaded/e.total);
    progress.innerHTML = p + "% загружено";
  }
}
```

Кроме рассмотренных выше полезных событий отслеживания загрузки, спецификация XHR 2 описывает события, позволяющие отслеживать выгрузку HTTP-запроса. В браузерах, поддерживающих XHR 2, объект XMLHttpRequest имеет свойство upload. Его значение — объект, определяющий метод addEventListener() и полный набор свойств событий прогресса, таких как onprogress и onload.

Обработчики событий upload используются так же, как и обычные обработчики прогресса. В объекте x типа XMLHttpRequest можно установить свойство x.onprogress для отслеживания прогресса загрузки или x.upload.onprogress для отслеживания прогресса выгрузки.

Кроссдоменные запросы

Согласно правилу ограничения домена (см. главу 6) объект XMLHttpRequest обычно может передавать запросы только на сервер, с которого загружен документ. Это ограничение закрывает ряд брешей в системе безопасности, но оно же предотвращает ряд полезных применений кроссдоменных запросов. Кроссдоменные URL-адреса можно использовать в элементах <form> и <iframe>, в результате чего браузер отобразит результирующий кроссдоменный документ. Но вследствие правила ограничения домена браузер не позволит исходному сценарию просматривать содержимое кроссдоменного документа, т.е. на экране оно видно,

но для сценария недоступно. При использовании объекта XMLHttpRequest содержимое документа можно получить из свойства responseText, но то же самое правило ограничения домена не разрешит объекту XMLHttpRequest передать кроссдоменный запрос. Обратите внимание на то, что элемент <script> не подвержен правилу ограничения домена. Он может загрузить и выполнить любой сценарий независимо от адреса, где он находится. Далее будет показано, что в технологии Ajax эта свобода кроссдоменных запросов делает элемент <script> привлекательной альтернативой объекту XMLHttpRequest.

Спецификация XHR 2 разрешает кроссдоменные запросы к определенным веб-сайтам, указанным в заголовках CORS HTTP-ответов. Если браузер поддерживает заголовки CORS объекта XMLHttpRequest и запрашиваемый сайт разрешает кроссдоменные запросы с заголовками CORS, правило ограничения домена существенно смягчается, и сценарий может передавать кроссдоменные запросы.

Технология JSONP: HTTP-запросы в элементе <script>

Для определенных типов содержимого элемент <script> можно использовать как полезную альтернативу объекту XMLHttpRequest. Вставьте атрибут src в дескриптор <script>, и браузер сгенерирует HTTP-запрос, загружающий указанный ресурс. Элементы <script> полезны в технологии Ajax тем, что они не подчиняются правилу ограничения домена, в результате чего их можно использовать для получения данных от других серверов.

Методика использования элемента <script> для передачи данных Ajax называется технологией JSONP. Она работает, когда тело HTTP-ответа закодировано в формате JSON.

Предположим, существует служба, обрабатывающая запросы GET и возвращающая данные в формате JSON. Документы с этого же сервера могут обрабатывать полученные данные с помощью объекта `XMLHttpRequest` и метода `JSON.parse()`. Если на сервере включить поддержку заголовков CORS, то в новых браузерах кроссдоменные документы также смогут пользоваться службой с помощью объекта `XMLHttpRequest`. Однако старые браузеры, не поддерживающие заголовки CORS, имеют доступ к службе только через элемент `<script>`. Тело документа в формате JSON по определению является правильным кодом JavaScript, и браузер выполнит его при получении с сервера. Обработка данных JSON приводит к их декодированию, но это по-прежнему всего лишь данные, которые ничего не делают.

Суффикс "P" в слове "JSONP" означает "Padding" (подкладка). При вызове через элемент `<script>` служба должна заключить ответ в скобки и добавить в качестве префикса имя функции JavaScript. Запрос JSON передает следующие данные.

```
[1, 2, {"Обувь": "туфли"}]
```

Ответ JSONP выглядит так.

```
handleResponse(
[1, 2, {"Обувь": "туфли"}]
)
```

Как тело элемента `<script>`, ответ с подкладкой не делает ничего полезного: он вычисляет данные в формате JSON (а это всего лишь выражение JavaScript) и передает их функции `handleRqsponse()`, которая, как предполагается в документе, должна сделать что-то полезное с данными.

Чтобы все это сработало, нужно каким-то образом сообщить службе, что запрос передается элементом `<script>` и необходимо прислать ответ JSONP, а не JSON. Это мож-

но сделать, добавив параметр запроса в строку URL, например добавив `?json` или `&json`.

На практике службы, поддерживающие JSONP, не требуют от клиентских устройств использовать какое-либо определенное имя функции, такое как `handleResponse`. В службах, чтобы разрешить клиенту задать имя функции, используется значение параметра запроса. Затем это имя используется в ответе в качестве подкладки. В листинге 13.3 для задания имени функции обратного вызова используется параметр запроса `jsonp`.

В листинге 13.3 приведено определение функции `getJSONP()`, создающей запрос JSONP. Данный пример довольно коварный, и вы должны хорошо понимать, как он работает. Во-первых, обратите внимание на то, как код создает элемент `<script>`, устанавливает URL-адрес и вставляет их в документ. Процесс вставки запускает HTTP-запрос. Во-вторых, обратите внимание на то, как код создает новую внутреннюю функцию обратного вызова для каждого запроса, сохраняя функцию как свойство `getJSONP()`. И наконец, проанализируйте, как функция обратного вызова "зачищает" код, удаляя элемент `<script>` и саму себя.

Листинг 13.3. Создание запроса JSONP с помощью элемента `<script>`

```
// Создание запроса по заданному URL-адресу и
// передача обработанных данных ответа в функцию
// обратного вызова. Добавление параметра
// запроса "jsonp" в URL для задания имени
// функции обратного вызова для ответа.
function getJSONP(url, callback) {
  // Создание уникального имени функции обратного
  // вызова для данного запроса. Имя будет
  // свойством этой функции.
  var cbnum = "cb" + getJSONP.counter++;
  var cbname = "getJSONP." + cbnum;
  // Добавление имени функции обратного вызова
  // в URL-строку запроса
```

```
  if (url.indexOf("?") === -1)
    url += "?jsonp=" + cbname;
  else
    url += "&jsonp=" + cbname;

  // Создание элемента <script>
  var script = document.createElement("script");

  // Определение функции обратного вызова
  getJSONP[cbnum] = function(response) {
    try {
      callback(response); // Обработка ответа
    }
    finally { // Очистка (даже в случае ошибки)
      delete getJSONP[cbnum];
      script.parentNode.removeChild(script);
    }
  };

  // Запуск HTTP-запроса
  script.src = url;
  document.body.appendChild(script);
}

// Счетчик, используемый для генерации уникальных
// имен функций обратного вызова
getJSONP.counter = 0;
```

Безопасность сценариев

Чтобы использовать элемент <script> для передачи данных Ajax, нужно разрешить веб-странице выполнять код JavaScript, полученный с удаленного сервера. Это означает, что описанные методики нельзя применять при работе с ненадежным сервером. Используя их при работе с надежным сервером, учитывайте, что, если хакер его взломает, он получит контроль над вашей страницей, сможет выполнить любой код и отобразить любое содержимое, причем оно будет выглядеть так, как будто получено с вашего сайта.

В последнее время на реальных веб-сайтах часто используются сценарии сторонних производителей, особенно для внедрения на страницу рекламы и презентационных элементов. Использование дескриптора `<script>` как транспортного средства Ajax для взаимодействия с надежными веб-службами — не более опасная операция, чем использование сценариев сторонних производителей.

Протокол Server-Sent Event

При обычном взаимодействии по протоколу HTTP с помощью средств XHR или дескриптора `<script>` клиент запрашивает или извлекает данные с сервера, когда в них возникает необходимость. Но есть и другой стиль сетевого взаимодействия посредством протокола HTTP — модель Comet: клиент и сервер устанавливают HTTP-соединение и оставляют его открытым. Это позволяет серверу передавать данные клиенту без предварительного запроса.

Реализовать модель Comet средствами XHR довольно тяжело, но стандарт Server-Sent Events, появившийся в HTML5, определяет простую спецификацию EventSource API, облегчающую получение сообщений, сгенерированных сервером. Для обработки серверных событий достаточно передать URL-адрес конструктору `EventSource()`, а затем начать прослушивать события сообщений через полученный объект.

```
var ticker = new EventSource("stockprices.php");
ticker.onmessage = function(e) {
  var type = e.type;
  var data = e.data;
  // Здесь нужно обработать событие
}
```

Объект, ассоциированный с событием сообщения, имеет свойство `data`, содержащее строку, которую сервер передал в ответ на событие. Кроме того, объект события имеет свойство `type`. По умолчанию оно равно `"message"`, но источник события может задать другую строку. Один обработчик события `onmessage` получает все события серверного источника и при необходимости может диспетчеризовать их на основе свойства `type`.

Протокол Server-Sent Events довольно прост. Клиент инициирует соединение с сервером (при создании объекта `EventSource`), а сервер поддерживает соединение открытым. При возникновении события сервер записывает строку текста в соединение. Передаваемое событие может выглядеть, например, следующим образом.

```
event: bid // Тип события
data: GOOG // Установка свойства data
data: 999  // Добавление строки и данных
           // Пустая строка создает событие
```

Протокол WebSocket

Все описанные выше технологии сетевого взаимодействия основаны на протоколе HTTP. Это означает, что все они ограничены фундаментальными особенностями HTTP: это протокол без учета состояний на основе запросов клиента и ответов сервера. Протокол HTTP узко специализированный. Он предназначен главным образом для получения веб-страниц и других ресурсов. Более общие сетевые протоколы определяют долгоживущие соединения и дуплексную систему сообщений посредством TCP-сокетов. Предоставлять клиентскому коду JavaScript доступ к низкоуровневым TCP-сокетам небезопасно, однако библиотека WebSocket API реализует безопасную альтернативу: она позволяет клиентскому коду создавать

дуплексные сокетные соединения с сервером, поддерживающим протокол WebSocket. Это существенно облегчает решение многих сетевых задач.

Функции WebSocket API легко применять. Сначала нужно создать сокет с помощью конструктора `WebSocket()`.

```
var s = new WebSocket("ws://ws.example.com/resource");
```

Аргумент конструктора `WebSocket()` представляет собой URL-адрес с протоколом `ws://` (или `wss://` для безопасных соединений, подобно `https://`). URL-адрес задает хост и маршрут ресурса и может задавать порт (в WebSocket по умолчанию используются те же порты, что в HTTP или HTTPS).

Создав сокет, нужно зарегистрировать для него обработчик события.

```
s.onopen = function(e) { /* Открытие сокета */ };
s.onclose = function(e) { /* Закрытие сокета */ };
s.onerror = function(e) { /* Ошибка! */ };
s.onmessage = function(e) {
  var m = e.data; /* Сервер передал сообщение */
};
```

Чтобы передать сообщение на сервер, нужно вызвать метод `send()` сокета.

```
s.send("Привет, сервер!");
```

Когда сеанс работы с сервером завершен, нужно закрыть сокет, вызвав его метод `close()`.

Взаимодействие посредством протокола WebSocket является двунаправленным. Когда соединение WebSocket установлено, клиент и сервер могут обмениваться сообщениями в любой момент, и это не обязательно должны быть запросы и ответы.

Хранение данных на стороне клиента

В веб-приложениях API-функции браузера можно использовать для локального хранения данных на компьютере пользователя. Например, веб-приложение может сохранять пользовательские настройки и данные о состоянии сеанса, чтобы при следующем посещении страницы возобновить работу точно с того места, в котором она была прервана. Хранилища на стороне клиента обычно раздельные: страницы одного сайта не могут читать данные, сохраненные страницами другого сайта. Но две страницы одного сайта могут иметь общее хранилище и применять его в качестве механизма взаимодействия. Данные формы, введенные на одной странице, могут быть отображены в таблице на другой странице. Веб-приложение может задать время жизни сохраняемых данных. Например, можно задать хранение данных только до тех пор, пока не будет закрыто окно или браузер. Можно также задать сохранение данных на жестком диске, чтобы они были доступны через месяц или год. В этой главе рассматриваются два механизма хранения данных на стороне клиента: новая спецификация Web Storage API и традиционные файлы "cookie".

Хранение, безопасность и конфиденциальность

Браузеры часто предлагают пользователю запомнить введенный им пароль и сохраняют его безопасным способом в зашифрованном виде на диске. Однако ни в одной из технологий хранения данных, рассматриваемых в данной главе, шифрование не применяется. Браузер сохраняет все на жестком диске пользователя в незашифрованном виде. Следовательно, сохраняемые данные доступны для любопытных пользователей, имеющих общий доступ к компьютеру, и зловредных программ, которые могут выполняться на компьютере. По этой причине ни один из рассматриваемых в данной главе способов хранения данных на стороне клиента нельзя применять для сохранения паролей, сведений о банковских счетах и другой важной информации.

Учитывайте также, что многие пользователи не доверяют сайтам, в которых используются файлы "cookie" или другие механизмы хранения на стороне клиента, позволяющие каким-либо образом отслеживать действия клиента. Применяйте рассматриваемые в данной главе методы хранения только для того, чтобы пользователям было удобнее работать на сайте. Избегайте нарушения конфиденциальности пользовательских данных. Если слишком много сайтов будут злоупотреблять данными на стороне клиента, то пользователи отключат хранилища или будут часто их очищать, в результате чего такое хранение потеряет смысл.

Свойства `localStorage` и `sessionStorage`

Браузеры, реализующие спецификацию Web Storage, определяют два свойства объекта `Window`, предназначенных для хранения данных: `localStorage` и `sessionStorage`. Оба свойства ссылаются на объект `Storage` — постоянный

ассоциативный массив, связывающий строковые ключи со строковыми значениями. Объекты Storage работают так же, как и обычные объекты JavaScript. Код может присвоить строку свойству объекта, и браузер сохранит эту строку для дальнейшего использования. Разница между свойствами localStorage и sessionStorage заключается во времени жизни и области видимости, т.е. в том, как долго хранятся данные и в каких местах кода они доступны.

Далее будут подробно обсуждаться вопросы времени жизни и области видимости хранилищ. Но сначала рассмотрим несколько примеров. В приведенном ниже коде используется свойство localStorage, но он может работать и со свойством sessionStorage.

```
// Извлечение сохраненного значения
var name = localStorage.username;
// Эквивалентная инструкция в виде массива
name = localStorage["username"];
if (!name) { // Если имени нет, создаем его
  name = prompt("Ваше имя?");
  localStorage.username = name;
}
// Проход по парам "имя-значение"
for(var key in localStorage) {
  var value = localStorage[key];
}
```

В объектах Storage определены методы сохранения, извлечения, последовательного просмотра и удаления данных. Они будут рассмотрены далее.

Согласно проекту спецификации Web Storage сохранять можно как структурированные данные (объекты и массивы), так и примитивные значения, такие как встроенные типы дат, регулярные выражения и даже объекты File. Однако на момент написания этой книги браузеры могут сохранять только строки. Если нужно сохранять

и считывать другие типы данных, их нужно кодировать и декодировать.

```
// Сохраняемые числа автоматически преобразуются
// в строки. После извлечения их нужно
// преобразовать в числа
localStorage.x = 10;
var x = parseInt(localStorage.x);

// Преобразование даты в строку
localStorage.lastRead = (new Date()).toUTCString();
// Преобразование строки в дату при извлечении
var last = new Date(Date.parse(localStorage.lastRead));

// Используйте JSON для преобразования
// объектов и массивов в строку
localStorage.data = JSON.stringify(data);
var data = JSON.parse(localStorage.data);
```

Время жизни и область видимости хранилища

Как уже упоминалось, свойства `localStorage` и `sessionStorage` различаются временем жизни и областью видимости хранилища. Данные, записываемые через свойство `localStorage`, хранятся постоянно. Они будут находиться на жестком диске до тех пор, пока их не удалит веб-приложение или пока пользователь не попросит браузер (посредством специальных элементов интерфейса) удалить их.

Область видимости свойства `localStorage` определяется правилом ограничения домена. Как было показано в главе 10, происхождение документа определяется его протоколом, именем хоста и портом. Все приведенные ниже URL-адреса имеют разное происхождение.

```
http://www.example.com
https://www.example.com      // Разные протоколы
http://static.example.com    // Разные хосты
http://www.example.com:8000 // Разные порты
```

Все документы общего происхождения имеют общее свойство localStorage (независимо от происхождения сценария, обращающегося к значениям localStorage). Все такие документы могут читать и записывать данные друг друга. Но документ другого происхождения не может читать или записывать эти данные (даже если в нем выполняется сценарий с того же сервера).

Область видимости свойства localStorage ограничена типом браузера. Например, если посетить сайт сначала с помощью браузера Firefox, а затем — браузера Chrome, то любые данные, сохраненные при первом посещении, при втором посещении не будут доступны.

Данные, сохраненные через свойство sessionStorage, имеют то же время жизни, что и верхнеуровневое окно или вкладка браузера, в которой выполняется сценарий, сохранивший данные. Когда окно или вкладка закрывается (но не сворачивается), все данные, сохраненные через свойство sessionStorage, удаляются. Однако некоторые современные браузеры могут восстанавливать недавно закрытые вкладки и последний сеанс просмотра. Одновременно восстанавливаются и свойства sessionStorage, поэтому время жизни данных может быть бо́льшим.

Как и localStorage, свойство sessionStorage имеет область видимости, определяемую происхождением документа. Документы другого происхождения не имеют доступа к значениям свойства sessionStorage. Однако область видимости свойства sessionStorage ограничена также окном. Если в браузере открыты две вкладки, отображающие документы одного происхождения, то в этих вкладках используются отдельные области видимости свойств sessionStorage. Это значит, что сценарии, выполняющиеся в одной вкладке, не могут читать или записывать данные, записанные сценариями другой вкладки,

даже если в обеих вкладках открыта одна и та же страница и выполняются одни и те же сценарии.

Учитывайте, что область видимости свойства `sessionStorage` ограничена только окном верхнего уровня. Если во вкладке есть два элемента `<iframe>` и в этих фреймах отображаются два документа общего происхождения, то сценарии фреймов будут иметь общее свойство `sessionStorage`.

Встроенные функции хранения данных

Свойства `localStorage` и `sessionStorage` часто используются так, как будто это обычные объекты JavaScript: установите свойство, чтобы сохранить строку, и прочитайте свойство, чтобы извлечь данные. Но в этих объектах определены более формальные методы. Для сохранения значения передайте имя и значение методу `setItem()`. Чтобы извлечь значение, передайте имя методу `getItem()`. Чтобы удалить пару "имя–значение", передайте имя методу `removeItem()`. В большинстве браузеров для удаления пары "имя–значение" можно также использовать оператор `delete`, как и для удаления обычного объекта, но такой способ в IE8 не работает. Для удаления всех сохраненных значений вызовите метод `clear()` без аргументов. Количество имен сохраненных значений хранится в свойстве `length`. Для циклического обхода значений определите переменную счетчика, изменяемую от 0 до `length-1`, и передайте ее методу `key()`. Ниже приведен пример использования свойства `localStorage`. Этот же код может работать и со свойством `sessionStorage`.

```
localStorage.setItem("x", 1); // Сохранение "x"
localStorage.getItem("x");    // Извлечение "x"

// Проход в цикле по парам "имя-значение"
for(var i = 0; i < localStorage.length; i++) {
  // Получение имени пары номер i
```

```
  var name = localStorage.key(i);
  // Получение значения пары
  var value = localStorage.getItem(name);
}

localStorage.removeItem("x"); // Удаление "x"
localStorage.clear();         // Удаление всех данных
```

События хранилища

При любом изменении данных, хранящихся в свойствах `localStorage` и `sessionStorage`, браузер генерирует событие `storage` во всех объектах `Window`, в которых видны эти данные (но не в объекте окна, который внес изменение). Если в браузере открыты две вкладки со страницами общего происхождения и одна из них хранит значение в свойстве `localStorage`, то другая страница получит событие `storage`. Как вы помните, область видимости свойства `sessionStorage` совпадает с верхнеуровневым окном, поэтому события `storage` генерируются, только если в изменение вовлечены фреймы. Кроме того, событие `storage` генерируется, только когда произошло фактическое изменение. Присвоение элементу хранилища его существующего значения или удаление несуществующего элемента не приведет к возникновению события.

Регистрация обработчика события `storage` выполняется с помощью метода `addEventListener()` (в IE он называется `attachEvent()`). В большинстве браузеров можно также присвоить обработчик свойству `onstorage` объекта `Window`, но на момент написания книги браузер Firefox не поддерживает данное свойство.

Объект события `storage` имеет пять важных свойств (к сожалению, в IE8 они не поддерживаются).

- **key** — имя или ключ устанавливаемого или удаляемого элемента. После вызова метода `clear()` это свойство равно `null`.
- **newValue** — новое значение элемента или `null`, если вызван метод `removeItem()`.
- **oldValue** — прежнее значение элемента (измененяемое или удаляемое) или `null` при вставке элемента.
- **storageArea** — равно либо свойству `localStorage`, либо свойству `sessionStorage` целевого объекта `Window`.
- **url** — строка URL-адреса документа, сценарий которого выполнил изменение хранилища.

Свойство `localStorage` и событие `storage` могут служить механизмом широковещательного распространения сообщений всем окнам браузера, в которых отображен данный сайт. Например, если пользователь попросит сайт остановить анимацию, сайт может сохранить этот запрос в свойстве `localStorage` таким образом, что он будет учитываться в других окнах. Сохранив запрос пользователя, браузер сгенерирует событие, позволяющее другим окнам увидеть и выполнить его. Еще пример: представьте себе сетевое приложение для редактирования изображений, отображающее панель инструментов в отдельном окне. Когда пользователь выбирает инструмент, приложение использует свойство `localStorage` для сохранения текущего состояния и генерации для других окон извещения о том, что выбран новый инструмент.

Файлы "cookie"

Файл "cookie" — это небольшой именованный фрагмент данных, хранящийся в браузере и ассоциированный

с конкретной веб-страницей или сайтом. Изначально файлы "cookie" были предназначены для программирования на стороне сервера. На самом низком уровне они реализованы как расширение протокола HTTP. Файлы "cookie" автоматически перемещаются между браузером и сервером, в результате чего серверные сценарии могут читать и редактировать файлы "cookie", хранящиеся на стороне клиента. В этом разделе показано, как сценарии на стороне клиента могут манипулировать файлами "cookie" с помощью свойства `cookie` объекта `Document`.

Программный интерфейс для работы с файлами "cookie" довольно старый и по этой причине поддерживается везде. К сожалению, он весьма сложный. Методы в нем не используются, а все операции с файлами "cookie" (чтение, установка и удаление) выполняются посредством чтения и записи свойства `cookie` объекта `Document` с помощью специально отформатированных строк. Время жизни и область видимости каждого файла "cookie" можно задавать индивидуально с помощью атрибутов. Эти атрибуты также определяются с помощью специально отформатированных строк, присвоенных тому же свойству `cookie`.

В следующих разделах рассматриваются атрибуты файла "cookie", задающие его время жизни и область видимости, а также демонстрируется установка и чтение файлов "cookie" в коде JavaScript.

Атрибуты записи "cookie": время жизни и область видимости

Кроме имени и значения у каждой записи "cookie" есть необязательные атрибуты, определяющие время жизни и область видимости. По умолчанию записи "cookie" непостоянны. Их значения хранятся на протяжении сеанса

браузера и теряются, когда пользователь закрывает браузер. Обратите внимание на то, что это немного отличается от правил для времени жизни данных sessionStorage: файлы "cookie" не ассоциированы с одним окном, и их время жизни совпадает с временем жизни процесса браузера, а не окна. Если нужно хранить файл "cookie" дольше сеанса браузера, то необходимо сообщить браузеру максимальное время хранения (в секундах) в атрибуте max-age. Если задано время жизни, браузер сохраняет файл "cookie" и удаляет его только по истечении времени жизни.

Область видимости файла "cookie" определяется происхождением документа, как и область видимости свойств localStorage и sessionStorage, а также маршрутом документа. Область видимости файла "cookie" можно конфигурировать с помощью атрибутов path и domain. По умолчанию файл "cookie" ассоциирован с веб-страницей, создавшей его, и доступен на этой и любой другой странице в том же каталоге и его подкаталогах. Например, если страница http://www.example.com/catalog/index.html создает файл "cookie", то он видим также для страниц http://www.example.com/catalog/order.html и http://www.example.com/catalog/widgets/index.html, но не для страницы http://www.example.com/about.html.

Чаще всего необходимо именно такое поведение, установленное по умолчанию. Но иногда желательно использовать файл "cookie" по всему сайту независимо от того, на какой странице он создан. Например, если пользователь вводит в форму свой электронный адрес, то желательно сохранить его как используемый в следующий раз по умолчанию на всех страницах сайта, в том числе и в другой форме.

Для этого необходимо задать атрибут path записи "cookie". Тогда любая веб-страница на этом же сервере, адрес которой начинается с указанного в path префикса,

будет иметь доступ к данному файлу "cookie". Например, если файл "cookie" установлен страницей `http://www.example.com/catalog/widgets/index.html` и атрибут `path` имеет значение `/catgalog`, то файл "cookie" виден также странице `http://www.example.com/catalog/order.html`. Если же атрибут `path` равен `/`, то файл "cookie" виден любой странице на сервере `http://www.example.com`.

Установка атрибута `path` равным `/` создает для файла "cookie" такую же область видимости, как у свойства `localStorage`, и, кроме того, приказывает браузеру передавать на сервер имя и значение записи "cookie" при каждом запросе к данному сайту.

По умолчанию файлы "cookie" имеют область видимости, определяемую происхождением документа. Однако на крупных сайтах часто желательно, чтобы файлы "cookie" были общими для вложенных доменов. Например, если для сервера `order.example.com` необходимо прочитать файл "cookie" вложенного домена `catalog.example.com`, то полезным будет атрибут `domain`. Если файл "cookie", созданный страницей с домена `catalog.example.com`, присваивает атрибуту `path` значение `/` и атрибуту `domain` — значение `.example.com`, то этот файл будет доступен для всех страниц доменов `catalog.example.com` и `orders.example.com`, а также на любом сервере домена `example.com`. Если атрибут `domain` не установлен, по умолчанию он равен имени хоста на сервере, обслуживающем страницу. Важно отметить, что установить для файла "cookie" домен, отличный от домена сервера, невозможно.

И наконец, последний атрибут `secure` определяет, как файл "cookie" может передаваться по сети. По умолчанию файл "cookie" считается опасным. Это означает, что он передается посредством обычного соединения HTTP. Но если файл "cookie" отмечен как безопасный, он может пе-

редаваться только посредством соединения HTTPS или посредством другого безопасного протокола.

Создание записей "cookie"

Чтобы ассоциировать значение временной записи "cookie" с текущим документом, нужно присвоить свойству `cookie` строку следующего формата:

```
имя=значение
```

Это можно сделать так.

```
var v = encodeURIComponent(document.lastModified);
document.cookie = "version=" + v;
```

При следующем чтении свойства `cookie` сохраненная пара "имя–значение" будет включена в список записей "cookie" документа. Записи "cookie" не могут содержать двоеточия, запятые и пробелы. По этой причине для кодирования значения рекомендуется использовать глобальную функцию JavaScript `encodeURIComponent()`.

Простой файл "cookie" с парой "имя–значение" существует на протяжении текущего сеанса браузера и теряется, когда пользователь закрывает браузер. Для создания файла "cookie", существующего дольше сеанса, нужно задать время жизни в секундах в атрибуте `max-age`. Для этого нужно присвоить свойству `cookie` строку следующего формата:

```
имя=значение; max-age=секунды
```

Приведенная ниже функция устанавливает файл "cookie" с атрибутом `max-age`.

```
// Сохранение пары "имя-значение" как записи "cookie";
// кодирование значения с помощью функции
// encodeURIComponent(), чтобы заменить двоеточия,
// запятые и пробелы. Если параметр daysToLive
// числовой, функция устанавливает атрибут
// max-age. При передаче нуля запись удаляется
```

```
function setCookie(name, value, daysToLive) {
  var cookie = name + "=" + encodeURIComponent(value);
  if (typeof daysToLive === "number")
    cookie += "; max-age=" + (daysToLive*60*60*24);
  document.cookie = cookie;
}
```

Атрибуты `path`, `domain` и `secure` устанавливаются аналогично — путем добавления строк в указанных ниже форматах к строке "cookie" перед записью в свойство `cookie`.

```
path=маршрут

domain=домен

secure
```

Чтобы изменить запись "cookie", установите ее повторно. При этом нужно повторно установить также все атрибуты записи. Можно задать другое время жизни с помощью нового атрибута `max-age`.

Для удаления записи "cookie" установите ее повторно с тем же именем и атрибутами, но с произвольным значением и с атрибутом `max-age`, равным нулю.

Чтение записей "cookie"

При использовании свойства `cookie` в выражении JavaScript возвращается строка, содержащая все записи "cookie" текущего документа. Строка состоит из списка пар *имя=значение*, разделенных точками с запятой и пробелами. Значение не содержит атрибуты, присвоенные записи "cookie". Обычно при использовании свойства `document.cookie` его разбивают на отдельные пары "имя–значение" с помощью метода `split()`.

После извлечения записи "cookie" из свойства `cookie` ее нужно интерпретировать на основе формата, приме-

ненного при создании записи. Например, можно передать значение записи "cookie" методам `decodeURIComponent()` и `JSON.parse()`.

В листинге 14.1 приведена функция `getCookie()`, которая выполняет синтаксический разбор свойства `document.cookie` и возвращает объект со свойствами, содержащими имя и значение записи "cookie".

Листинг 14.1. Синтаксический анализ свойства `document.cookie`

```
// Возвращение записи "cookie" документа как объекта,
// состоящего из пар "имя-значение"
// Предполагается, что значения были
// закодированы с помощью метода
// encodeURIComponent().
function getCookies() {
  var cookies = {};          // Возвращаемый объект
  var all = document.cookie; // Все записи "cookie"
  if (all === "")            // Если свойство пустое,
    return cookies;          // возвращаем пустой объект
  // Разбиение строки на пары
  var list = all.split("; ");
  // Проход в цикле по парам
  for(var i = 0; i < list.length; i++) {
    var cookie = list[i];
    // Разбиение каждой пары
    var p = cookie.indexOf("=");
    var name = cookie.substring(0,p);
    var value = cookie.substring(p+1);
    // Сохранение имени и декодированного значения
    cookies[name] = decodeURIComponent(value);
  }
  return cookies;
}
```

Ограничения файлов "cookie"

Файлы "cookie" предназначены для сохранения в браузере небольшого объема данных серверными сценария-

ми. Эти данные передаются на сервер при каждом запросе соответствующего URL-адреса. Стандарт, определяющий файлы "cookie", поощряет производителей браузеров разрешать неограниченное количество файлов неограниченного размера, но не требует от браузеров сохранят более 300 записей (по 20 файлов "cookie" на сервер) или более 4 Кбайт данных на одну запись. На практике браузеры разрешают сохранять намного больше, чем 300 записей, но некоторые браузеры накладывают ограничение на размер записи: он должен быть не более 4 Кбайт.

Предметный указатель

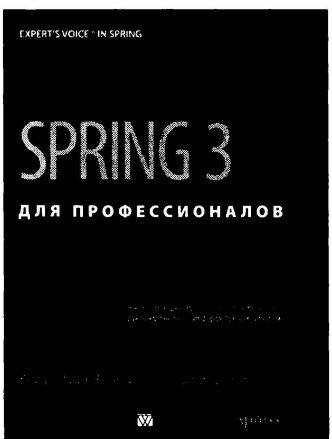
П

Р

С

SPRING 3
ДЛЯ ПРОФЕССИОНАЛОВ

Кларенс Хо,
Роб Харроп



Гибкая, облегченная, с открытым кодом платформа Spring Framework продолжает занимать место лидирующей инфраструктуры для разработки приложений на Java для современных программистов и разработчиков. Она работает в тесной интеграции с другими гибкими и облегченными Java-технологиями с открытым кодом, такими как Hibernate, Groovy, MyBatis и т.д. В настоящее время Spring также может взаимодействовать с Java EE и JPA 2. Благодаря настоящей книге, вы изучите основы Spring, освоите ключевые темы, а также ознакомитесь с реальным опытом реализации в приложениях удаленной обработки, Hibernate и EJB. Помимо основ, вы узнаете, как использовать Spring Framework для построения различных уровней или частей корпоративного Java-приложения, в том числе транзакций, веб-уровня и уровня презентаций, развертывания и многого другого.

www.williamspublishing.com

ISBN 978-5-8459-1803-1

в продаже

C# 5.0
КАРМАННЫЙ СПРАВОЧНИК

Джозеф Албахари
Бен Албахари

www.williamspublishing.com

Книга является идеальным кратким справочником, позволяющим быстро найти исчерпывающую информацию по языку C# 5.0. В ней изложены все основные темы, касающиеся языка C# 5.0, как основы, так и более сложные темы, такие как перегрузка операторов, ограничения, ковариантности и контравариантность, итераторы, типы, допускающи нулевое значение, заимствование операторов, лямбда-выражения и замыкания. Кроме того, в книге изложена информаци о языке LINQ, начиная с последовательностей, отложенного выполнения и стандартных операторов запроса и заканчивая полным справочником по выражениям запроса. Описаны динамическое связывание и новые асинхронные функции в язы C# 5.0, а также вопросы, касающиеся небезопасного кода и указатели, собственные атрибуты, директивы препроцессоров и документация XML.

ISBN 978-5-8459-1820-8

в продаже

JAVA
РУКОВОДСТВО ДЛЯ НАЧИНАЮЩИХ
ПЯТОЕ ИЗДАНИЕ

Герберт Шилдт

www.williamspublishing.com

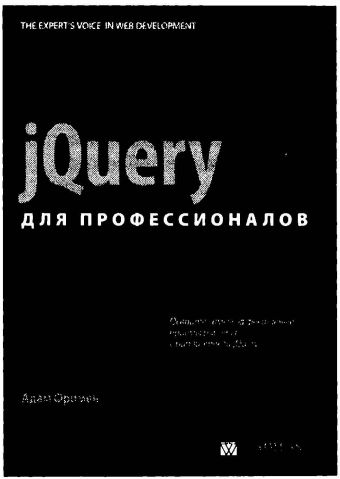
В этом учебном пособии для начинающих программировать на Java подробно рассмотрены все основные средства данного языка программирования: типы данных, операторы, циклы, классы, интерфейсы, методы, исключения, обобщения, пакеты, основные библиотеки классов, средства многопоточного программирования, потоки ввода-вывода, перечисления, апплеты и документирующие комментарии. Применение всех этих языковых средств Java на практике наглядно демонстрируется в небольших проектах для самостоятельного опробования. Книга снабжена массой полезных советов авторитетного автора и множеством примеров программ с подробными комментариями, благодаря которым они становятся понятными любому читателю независимо от уровня его подготовки. А для проверки прочности приобретенных знаний и навыков в конце каждой главы приводятся контрольные вопросы и задания. Книга рассчитана на широкий круг читателей, интересующихся программированием на Java.

ISBN 978-5-8459-1770-6

в продаже

jQUERY ДЛЯ ПРОФЕССИОНАЛОВ

Адам Фримен



В книге показано, как создавать профессиональные веб-приложения с меньшими усилиями и при меньшем размере кода. Вы изучите методы работы со встроенными и дистанционными данными, научитесь создавать функционально насыщенные интерфейсы для веб-приложений, а также познакомитесь с возможностями сенсорно-ориентированного фреймворка jQuery Mobile.

Основные темы книги:

- возможности и особенности библиотеки jQuery;
- применение базовых инструментов jQuery для улучшения HTML-документов, включения в них таблиц, форм и средств отображения данных;
- применение библиотеки jQuery UI для создания гибких и удобных в использовании веб-приложений;
- программирование различных элементов взаимодействия, как перетаскивание и вставка объектов, сортировка данных и сенсорная чувствительность;
- применение библиотеки jQuery Mobile при разработке сенсорно-ориентированных интерфейсов для мобильных устройств и планшетных компьютеров;
- расширение библиотеки jQuery путем создания собственных подключаемых модулей и виджетов.

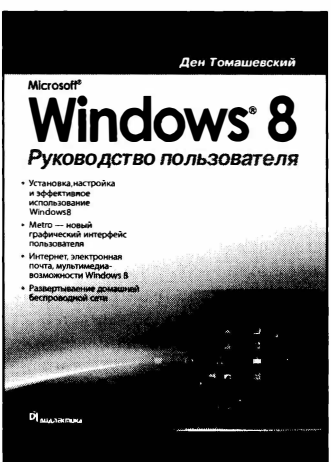
www.williamspublishing.com

ISBN 978-5-8459-1799-7

В продаже

MICROSOFT® WINDOWS® 8 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Ден Томашевский



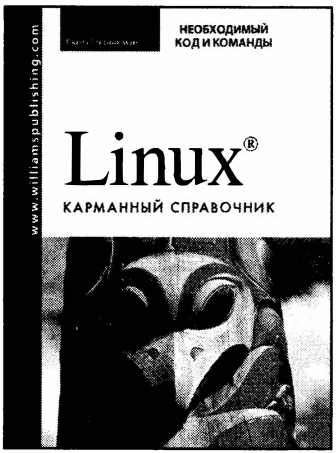
www.dialektika.com

В этой книге описывается последняя версия операционной системы от Microsoft — Windows 8. Рассказывается, как ее установить и настроить, какие возможности эта система предоставляет пользователю, в чем ее отличие от предыдущих версий и каковы особенности ее нового графического интерфейса Metro. Даются рекомендации по использованию стандартных программ и мультимедиа-возможностей Windows 8, подключению и работе в Интернете, организации домашней сети и настройке встроенного брандмауэра Windows. Книга рассчитана на пользователей любой квалификации и будет полезна как начинающим, так и достаточно опытным пользователям ПК, ноутбуков и планшетов.

ISBN 978-5-8459-1827-7 **в продаже**

LINUX
КАРМАННЫЙ СПРАВОЧНИК

Скотт Граннеман



www.williamspublishing.com

Данная книга представляет собой краткое пособие по основным командам операционной системы Linux. Читатель найдет в ней описание большинства команд, необходимых ему в повседневной работе.
В первых главах представлены те средства, с которых любой новичок начинает знакомство с неизвестной ему операционной системой. В них рассматриваются вопросы вывода на экран информации о каталогах, перехода из одного каталога в другой, создания каталогов, отображения содержимого файлов и т.д. По мере чтения книги материал усложняется. Читатель получает представление о работе с принтерами, правах доступа к файлам, архивировании и сжатии данных, поиске информации и др.
В книгу включено большое количество примеров, иллюстрирующих использование каждой описанной в ней команды.

ISBN 978-5-8459-1118-6 **в продаже**

Веб-разработка

O'REILLY®

Карманный справочник по JavaScript

JavaScript — популярнейший язык программирования, который уже более 15 лет применяется для написания сценариев интерактивных веб-страниц. В книге представлены самые важные сведения о синтаксисе языка и показаны примеры его практического применения. Несмотря на малый объем карманного издания, в нем содержится все, что необходимо знать для разработки профессиональных веб-приложений.

Главы 1–9 посвящены описанию синтаксиса последней версии языка JavaScript (спецификация ECMAScript 5).

- Типы данных, значения и переменные
- Инструкции, операторы и выражения
- Объекты и массивы
- Классы и функции
- Регулярные выражения

В главах 10–14 рассматриваются функциональные возможности языка наряду с моделью DOM и средствами поддержки HTML5.

- Взаимодействие кода JavaScript с окнами браузера
- Сценарии HTML-документов и элементы страницы
- Управление стилями и классами CSS посредством кода JavaScript
- Реагирование на события мыши и клавиатуры
- Взаимодействие с веб-серверами
- Хранение данных на компьютере пользователя

www.williamspublishing.com

oreilly.com

ISBN 978-5-8459-1830-7