По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-055-3, название «JavaScript. Подробное руководство, 4-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

JavaScript

The Definitive Guide

Fourth Edition

David Flanagan

O'REILLY®

JavaScript

Подробное руководство

Четвертое издание

Дэвид Флэнаган

Санкт-Петербург — Москва
2004

Дэвид Флэнаган

JavaScript. Подробное руководство, 4-е издание

Перевод Л. Фрейдина

Главный редактор	*А. Галунов*
Зав. редакцией	*Н. Макарова*
Научный редактор	*А. Королев*
Редактор	*В. Овчинников*
Корректура	*С. Беляева, С. Журавина*
Верстка	*В. Овчинников*

Флэнаган Д.
JavaScript. Подробное руководство. – Пер. с англ. – СПб: Символ-Плюс, 2004. – 960 с., ил.
ISBN 5-93286-055-3

Четвертое издание бестселлера по JavaScript содержит полное описание базового языка JavaScript, а также традиционной и стандартизованной объектных моделей документа, реализованных в веб-броузерах. Примеры, включенные в книгу, можно использовать для решения распространенных задач, таких как проверка данных формы, работа с cookies и создание переносимой анимации DHTML. Части с III по V представляют собой подробные справочники по базовому API JavaScript, традиционному клиентскому API и стандартизованному API W3C DOM, в которых описываются все объекты, методы, свойства, конструкторы, константы, функции и обработчики событий этих API. Издание дополнено с учетом возможностей JavaScript 1.5 (ECMAScript v3) и содержит описание стандарта W3C DOM (Level 1 и Level 2), при этом для обратной совместимости сохранен материал по традиционному DOM Level 0.

Эта книга необходима каждому, кто пишет на JavaScript, независимо от его опыта. Она будет особенно полезна тем, кто работает с последними, соответствующими стандартам веб-броузерами, такими как Internet Explorer 6, Netscape 6 и Mozilla. Вебмастеры узнают, как применять JavaScript для построения динамических веб-страниц. Опытные разработчики смогут быстро приступить к написанию сложных программ.

ISBN 5-93286-055-3
ISBN 0-596-00048-0 (англ)

© Издательство Символ-Плюс, 2004
Authorized translation of the English edition © 2002 O'Reilly Media, Inc. This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

Все права на данное издание защищены Законодательством РФ, включая право на полное или частичное воспроизведение в любой форме. Все товарные знаки или зарегистрированные товарные знаки, упоминаемые в настоящем издании, являются собственностью соответствующих фирм.

Издательство «Символ-Плюс». 199034, Санкт-Петербург, 16 линия, 7,
тел. (812) 324-5353, edit@symbol.ru. Лицензия ЛП N 000054 от 25.12.98.
Налоговая льгота – общероссийский классификатор продукции
ОК 005-93, том 2; 953000 – книги и брошюры.

Подписано в печать 18.02.2004. Формат 70×100 1/16 . Печать офсетная.
Объем 60 печ. л. Тираж 2000 экз. Заказ N
Отпечатано с диапозитивов в Академической типографии «Наука» РАН
199034, Санкт-Петербург, 9 линия, 12.

Эта книга посвящается всем,
кто учит жить мирно и противостоит насилию.

Оглавление

Предисловие 11

1. Введение в JavaScript 16

1.1. Мифы о JavaScript 17
1.2. Версии JavaScript 18
1.3. Клиентский JavaScript 19
1.4. JavaScript в иных контекстах 21
1.5. Клиентский JavaScript: исполняемое содержимое веб-страниц 22
1.6. Возможности клиентского JavaScript 24
1.7. Безопасность в JavaScript 28
1.8. Пример: вычисление платежей по ссуде с помощью JavaScript 29
1.9. Структура этой книги 32
1.10. Изучение JavaScript 34

Часть I. Базовый JavaScript 37

2. Лексическая структура 39

2.1. Набор символов 39
2.2. Чувствительность к регистру 40
2.3. Символы-разделители и переводы строк 40
2.4. Необязательные точки с запятой 41
2.5. Комментарии 42
2.6. Литералы 42
2.7. Идентификаторы 43
2.8. Зарезервированные слова 43

3. Типы данных и значения 45

3.1. Числа 46
3.2. Строки 49
3.3. Логические значения 53
3.4. Функции 54
3.5. Объекты 56
3.6. Массивы 58
3.7. null 59
3.8. undefined 59
3.9. Объект Date 60

3.10. Регулярные выражения . . . 61
3.11. Объекты Error . . . 61
3.12. Объекты-обертки для базовых типов данных . . . 62

4. Переменные . . . 64

4.1. Типизация переменных . . . 64
4.2. Объявление переменных . . . 65
4.3. Область действия переменной . . . 66
4.4. Базовые и ссылочные типы . . . 69
4.5. Сборка мусора . . . 71
4.6. Переменные как свойства . . . 72
4.7. Еще об области действия переменных . . . 74

5. Выражения и операторы . . . 76

5.1. Выражения . . . 76
5.2. Обзор операторов . . . 77
5.3. Арифметические операторы . . . 81
5.4. Операторы равенства . . . 83
5.5. Операторы отношения . . . 86
5.6. Строковые операторы . . . 88
5.7. Логические операторы . . . 90
5.8. Поразрядные операторы . . . 91
5.9. Операторы присваивания . . . 93
5.10. Прочие операторы . . . 95

6. Инструкции . . . 101

6.1. Инструкции-выражения . . . 101
6.2. Составные инструкции . . . 102
6.3. if . . . 103
6.4. else if . . . 105
6.5. switch . . . 106
6.6. while . . . 109
6.7. do/while . . . 110
6.8. for . . . 110
6.9. for/in . . . 112
6.10. Метки . . . 113
6.11. break . . . 114
6.12. continue . . . 115
6.13. var . . . 116
6.14. function . . . 117
6.15. return . . . 119
6.16. throw . . . 119
6.17. try/catch/finally . . . 120
6.18. with . . . 123
6.19. Пустая инструкция . . . 124
6.20. Итоговая таблица инструкций JavaScript . . . 124

7. Функции 127

7.1. Определение и вызов функций 127
7.2. Функции как данные 132
7.3. Область видимости функции: объект вызова 134
7.4. Аргументы функции: объект Arguments 135
7.5. Свойства и методы функции 137

8. Объекты 140

8.1. Объекты и свойства 140
8.2. Конструкторы 143
8.3. Методы 144
8.4. Прототипы и наследование 146
8.5. Объектно-ориентированный JavaScript 151
8.6. Объекты как ассоциативные массивы 158
8.7. Свойства и методы класса Object 160

9. Массивы 166

9.1. Массивы и элементы массива 166
9.2. Методы массивов 170

10. Регулярные выражения 176

10.1. Определение регулярных выражений 176
10.2. Методы класса String для поиска по шаблону 186
10.3. Объект RegExp 189

11. Прочие вопросы программирования на JavaScript 192

11.1. Преобразования типов данных 192
11.2. Передача по значению и по ссылке 198
11.3. Сборка мусора 203
11.4. Лексический контекст и вложенные функции 206
11.5. Конструктор Function() и функциональные литералы 208
11.6. Несовместимость Netscape JavaScript 1.2 208

Часть II. Клиентский JavaScript 211

12. JavaScript в веб-броузерах 213

12.1. Среда веб-броузера 213
12.2. Встраивание JavaScript в HTML 217
12.3. Исполнение программ JavaScript 226

13. Окна и фреймы 232

13.1. Обзор объекта Window 232
13.2. Простые диалоговые окна 234
13.3. Строка состояния 237
13.4. Время запуска и интервалы 238

13.5. Обработка ошибок 240
13.6. Объект Navigator 240
13.7. Объект Screen 243
13.8. Методы управления окнами 243
13.9. Объект Location 248
13.10. Объект History 250
13.11. Работа с несколькими окнами и фреймами 252

14. Объект Document 259

14.1. Обзор объекта Document 260
14.2. Динамически генерируемые документы 264
14.3. Свойства цвета документа 269
14.4. Информационные свойства документа 270
14.5. Формы 270
14.6. Изображения 271
14.7. Ссылки 278
14.8. Якорные элементы 281
14.9. Апплеты 282
14.10. Вложенные данные 283

15. Формы и элементы форм 285

15.1. Объект Form 286
15.2. Определение элементов формы 288
15.3. Сценарии и элементы формы 292
15.4. Пример верификации формы 301

16. Сценарии и cookies 305

16.1. Обзор cookies 305
16.2. Сохранение cookie 307
16.3. Чтение cookie 309
16.4. Пример работы с cookie 310

17. Объектная модель документа 314

17.1. Обзор DOM 315
17.2. Использование базового DOM API 326
17.3. Совместимость DOM с Internet Explorer 4 345
17.4. Совместимость DOM с Netscape 4 347
17.5. Дополнительные методы: Traversal API и Range API 348

18. Каскадные таблицы стилей и Dynamic HTML 356

18.1. Стили и таблицы стилей в CSS 357
18.2. Позиционирование элемента с помощью CSS 365
18.3. Использование стилей в сценариях 375
18.4. DHTML в броузерах четвертого поколения 385
18.5. Другие DOM API для стилей и таблиц стилей 390

19. События и обработка событий 396

19.1. Базовая обработка событий 397
19.2. Развитые возможности обработки событий в DOM Level 2 . . . 408
19.3. Событийная модель Internet Explorer 425
19.4. Событийная модель Netscape 4 431

20. Приемы обеспечения совместимости 435

20.1. Совместимость с платформами и броузерами 436
20.2. Совместимость версий языка 441
20.3. Совместимость с броузерами, не поддерживающими JavaScript 445

21. Безопасность в JavaScript 448

21.1. JavaScript и безопасность 448
21.2. Ограничения в JavaScript 450
21.3. Политика общего происхождения 451
21.4. Зоны безопасности и подписанные сценарии 452

22. Совместное применение Java и JavaScript 454

22.1. Применение Java-апплетов в сценариях 455
22.2. Применение JavaScript из Java 456
22.3. Непосредственное использование классов Java 461
22.4. Типы данных LiveConnect 462
22.5. Преобразование данных в LiveConnect 467
22.6. Преобразование JavaObject в JavaScript 471
22.7. Преобразование данных из Java в JavaScript 473

Часть III. Справочник по базовому JavaScript 475

Часть IV. Справочник по клиентскому JavaScript 587

Часть V. Справочник по W3C DOM 729

Часть VI. Указатель классов, свойств, методов и обработчиков событий 895

Алфавитный указатель 912

Предисловие

Со времени публикации третьего издания этой книги в мире веб-программирования на JavaScript™ произошло много изменений, в том числе:

- Публикация второго и третьего изданий стандарта ECMA-262, обновивших ядро языка JavaScript. Были выпущены соответствующие версии интерпретаторов Netscape JavaScript и Microsoft JScript.
- Исходные тексты интерпретаторов JavaScript от Netscape (один написан на C, другой на Java™) выпущены в виде открытого исходного кода и доступны для всех, кто хочет встроить язык сценариев в свое приложение.
- World Wide Web Consortium (W3C) опубликовал две версии (два уровня) стандарта Document Object Model (DOM). Новейшие броузеры поддерживают этот стандарт (в разной степени) и позволяют клиентскому коду JavaScript взаимодействовать с содержимым документа и создавать сложные эффекты Dynamic HTML (DHTML). Широкую поддержку получили и другие стандарты W3C, такие как HTML 4, CSS1 и CSS2.
- Организация Mozilla, взяв за основу предоставленный Netscape исходный код, создала хороший броузер пятого поколения. Во время написания этой книги броузер Mozilla еще не дошел до уровня версии 1.0, но он уже достаточно зрелый для того, чтобы компания Netscape использовала базу кода Mozilla в качестве базы для своих броузеров версий 6.0 и 6.1.
- Microsoft Internet Explorer стал доминировать на настольных системах. Однако броузер Netscape/Mozilla по-прежнему важен для веб-разработчиков, в особенности по причине отличной поддержки веб-стандартов. Кроме того, как не менее важные надо рассматривать и второстепенные броузеры, такие как Opera (*http://www.opera.com*) и Konquerer (*http://www.konquerer.org*).
- Веб-броузеры (и интерпретаторы JavaScript) больше не ограничены настольными системами и работают даже на PDA и сотовых телефонах.

В итоге ядро языка JavaScript стало более зрелым. Язык стал стандартизованным и используется в более разнообразных средах, чем раньше. Коллапс доли рынка Netscape сделал возможным расширение ниши настольных веб-броузеров, а те из них, в которых реализована поддержка JavaScript, также стали доступными на платформах, отличных от настольных. Произошел явный, если не полный, переход к веб-стандартам. Частичная или полная ре-

ализация стандарта DOM в новейших броузерах дает веб-разработчикам давно ожидаемый независимый интерфейс API.

Что нового в 4-м издании

Это издание «JavaScript: The Definitive Guide» было тщательно переработано в свете только что описанных изменений. Самые важные из них включают полное описание JavaScript 1.5 и третьего издания стандарта ECMA-262, на котором он основан, а также полное описание стандарта DOM Level 2.

Фокус внимания смещен от конкретных реализаций JavaScript и броузеров (JavaScript 1.2, Netscape 4, IE 5 и т. д.) к документированию стандартов, на которых базируются (или должны базироваться) эти реализации. По причине множественности реализаций теперь практически невозможно описать в одной книге все возможности, фирменные расширения, индивидуальные особенности и ошибки всех реализаций, а любому разработчику – понять их. Книга ориентирована на спецификации, а не на реализации, что делает работу с ней проще, и если вы примете тот же подход, то ваш код JavaScript станет более переносимым и легким в сопровождении. Особое внимание в новом материале уделено стандартам по ядру JavaScript и DOM.

Другое существенное изменение в этом издании касается разбиения справочного раздела на три отдельные части. Материал по базовому JavaScript был отделен от материала по клиентскому JavaScript (часть IV) и помещен в самостоятельный раздел (часть III). Это сделано для удобства JavaScript-программистов, работающих с языком в среде, отличной от веб-броузера, и не интересующихся клиентским JavaScript.

Новый материал с описанием W3C DOM помещен в самостоятельный раздел (часть V), отдельно от уже известного материала по клиентскому JavaScript. Стандарт DOM определяет API, значительно отличающийся от «устаревшего» API традиционного клиентского JavaScript. В зависимости от целевого броузера разработчики выбирают тот или иной API и обычно не переключаются с одного на другой. Раздельное описание этих двух API также позволяет сохранить организацию справочного материала по клиентскому JavaScript (часть IV), удобную для читателей, знакомых с третьим изданием.

Чтобы освободить место для нового материала и не увеличивать чрезмерно объем книги, я убрал описания тривиальных свойств объектов, повторяющие материал разделов, посвященных объектам. Всем свойствам, которые требуют основательного рассмотрения, а также всем методам отведены специальные разделы. Кроме того, волшебники из O'Reilly заново оформили книгу; она стала меньше, а читать ее по-прежнему легко и удобно.

Типографские соглашения

В этой книге приняты следующие соглашения:

Шрифт OfficinaSansC

Применяется для выделения клавиш клавиатуры или элементов пользовательского интерфейса, таких как кнопка Back или меню Options.

Курсив

Обозначает первое вхождение термина. *Курсив* также применяется для адресов электронной почты, веб-сайтов, FTP-сайтов, имен файлов и каталогов, а также групп новостей. Кроме того, в этой книге *курсивом* выделяются имена Java-классов, чтобы их нельзя было спутать с именами JavaScript.

Моноширинный шрифт

Применяется для форматирования кода JavaScript, листингов HTML и вообще всего, что непосредственно набирается на клавиатуре при программировании.

Моноширинный курсив

Обозначает аргументы функций и элементы, которые в программе необходимо заменить на реальные значения.

Ошибки

Для улучшения будущих изданий и тиражей этой книги издательство O'Reilly & Associates просит сообщать о любых ошибках, неточностях, ложных или сбивающих с толку утверждениях и обычных опечатках. O'Reilly поддерживает веб-сайт, где можно найти список всех известных ошибок в этой книге:

http://www.oreilly.com/catalog/jscript4/errata/

На этой странице имеется ссылка на форму, заполнив которую можно отправить сообщение о найденной ошибке. Кроме того, можно сообщить об ошибках или задать вопросы по этой книге, послав письмо по электронной почте на адрес:

bookquestions@oreilly.com

Примеры в Сети

Примеры, приведенные в этой книге, можно загрузить с веб-сайта издательства, пройдя по ссылке *Examples* на странице:

http://www.oreilly.com/catalog/jscript4/

Комментарии и вопросы

Просьба присылать комментарии и вопросы по этой книге издателю:

O'Reilly & Associates, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

(800) 998-9938 (in the United States or Canada)

(707) 829-0515 (international/local)

(707) 829-0104 (fax)

Технические вопросы и комментарии по этой и другим книгам издательства O'Reilly можно также посылать по адресу:

bookquestions@oreilly.com

Дополнительная информация о книгах, конференциях, Resource Center и O'Reilly Network приведена на сайте издательства O'Reilly:

http://www.oreilly.com

Благодарности

Брендан Эйх (Brendan Eich) из Mozilla – один из создателей и главный новатор JavaScript. Я и другие программисты на JavaScript в огромном долгу перед ним за разработку JavaScript и за то, что в его сумасшедшем графике нашлось время для ответов на наши вопросы, причем он даже требовал еще вопросов. Кроме того, что Брендан терпеливо отвечал на мои многочисленные вопросы, он также прочитал первое и третье издания этой книги и дал очень полезные комментарии к ним.

Это руководство получило благословение первоклассных технических рецензентов, комментарии которых весьма способствовали улучшению этой книги. Валдемар Хорват (Waldermar Horwat) из Netscape рецензировал новый материал по JavaScript. Материал по W3C DOM прорецензирован Филиппом Ле Хегаре (Philippe Le Hegaret) из W3C, Питером-Паулем Кохом (Peter-Paul Koch), руководителем отдела программирования клиентских приложений в голландской компании Netlinq Framfab (*http://www.netlinqframfab.nl*), специализирующейся на интернет-консалтинге, разработке и внедрении цифровых технологий в Интернете и маркетинге, Диланом Шиманом (Dylan Schiemann) из SitePen (*http://www.sitepen.com*), а также независимым веб-разработчиком Джеффом Ятсом (Jeff Yates). Питер-Пауль и Джефф поддерживают веб-сайты, посвященные веб-дизайну и DOM – *http://www.xs4all.nl/~ppk/js/* и *http://www.pbwizard.com* соответственно. Джозеф Кесселман (Joseph Kesselman) из IBM Research ничего не рецензировал, но очень помог мне, отвечая на вопросы по W3C DOM.

Третье издание книги рецензировалось Бренданом Эйхом, Валдемаром Хорватом и Вайдуром Аппарао (Vidur Apparao) из Netscape, Германом Вентером

(Herman Venter) из Microsoft, двумя независимыми разработчиками JavaScript – Джеем Ходжесом (Jay Hodges) и Анжело Сиригосом (Angelo Sirigos). Дэн Шейфер (Dan Shafer) из CNET Builder.com выполнил некоторую предварительную работу по третьему изданию. Его материал не нашел применения в этом издании, но принадлежащие ему идеи и общие принципы принесли большую пользу. Норрис Бойд (Norris Boyd) и Скотт Фурман (Scott Furman) из Netscape, а также Скотт Айзекс (Scott Issacs) из Microsoft нашли время, чтобы поговорить со мной о будущем стандарте Document Object Model. И наконец, доктор Танкред Хиршманн (Dr. Tankred Hirschmann) показал глубокое понимание хитросплетений JavaScript 1.2.

Второе издание много выиграло от помощи и комментариев Ника Томпсона (Nick Thompson) и Ричарда Якера (Richard Yaker) из Netscape, доктора Шона Каценбергера (Dr. Shon Katzenberger), Ларри Салливана (Larry Sullivan) и Дэйва С. Митчелла (Dave C. Mitchell) из Microsoft, Линн Роллинс (Lynn Rollins) из R&B Communications. Первое издание рецензировалось Нилом Беркманом (Neil Berkman) из Bay Networks, Эндрю Шульманом (Andrew Schulman) и Терри Алленом (Terry Allen) из O'Reilly & Associates.

Эта книга также стала лучше благодаря многочисленным редакторам, которые над ней работали. Паула Фергюсон (Paula Ferguson) – редактор этого и третьего издания. Она тщательно просмотрела книгу, сделав ее более понятной и легкой для чтения. Френк Уиллисон (Frank Willison) редактировал второе издание, а Эндрю Шульман – первое.

И наконец, мои благодарности Кристи – как всегда и за очень многое.

Дэвид Флэнаган
сентябрь 2001

1

Введение в JavaScript

JavaScript – это облегченный, интерпретируемый язык программирования с объектно-ориентированными возможностями. Универсальное ядро языка было встроено в Netscape, Internet Explorer и другие веб-броузеры, а добавление в него объектов, представляющих окно веб-броузера и содержимое последнего, сделало язык привлекательным для веб-программирования. Эта клиентская версия JavaScript позволяет включать в веб-страницы исполняемое содержимое, то есть веб-страница более не обязана быть статическим HTML-кодом, а может включать в себя программы, взаимодействующие с пользователем, управляющие броузером и динамически создающие HTML.

С точки зрения синтаксиса базовый язык JavaScript напоминает C, C++ и Java такими программными конструкциями, как инструкция `if`, цикл `while` и оператор `&&`. Однако это подобие ограничивается синтаксической схожестью. JavaScript – это нетипизированный язык, то есть в нем не определяется тип переменных. Объекты в JavaScript больше похожи на ассоциативные массивы Perl,[1] чем на структуры C или объекты C++ или Java. Механизм объектно-ориентированного наследования JavaScript похож на механизм в малоизвестных языках Self и NewtonScript и сильно отличается от наследования в C++ и Java. Как и Perl, JavaScript – это интерпретируемый язык, и некоторые его инструменты, например регулярные выражения и средства работы с массивами, вдохновлены языком Perl.

В этой главе приводится краткий обзор JavaScript, объясняется, что JavaScript может делать и что не может, а также разоблачаются некоторые мифы. Здесь базовый язык JavaScript отделяется от встроенных и расширен-

---

[1] Ассоциативные массивы Perl также известны под названием «хеши». – *Примеч. науч. ред.*

ных версий, таких как клиентский JavaScript, встроенный в веб-броузеры, и серверный JavaScript, встроенный в веб-серверы Netscape (здесь описываются базовый и клиентский JavaScript). Кроме того, в данной главе на нескольких фрагментах клиентского JavaScript демонстрируется практическое веб-программирование.

1.1. Мифы о JavaScript

Вокруг JavaScript довольно много дезинформации и путаницы. Прежде чем двигаться дальше в изучении JavaScript, важно развенчать некоторые распространенные мифы, связанные с этим языком.

1.1.1. JavaScript – это не Java

Одно из наиболее распространенных заблуждений о JavaScript состоит в том, что он представляет собой упрощенную версию Java, языка программирования от компании Sun Microsystems. Кроме некоторой синтаксической схожести и способности предоставлять исполняемое содержимое для веб-броузеров, эти два языка между собой ничто не связывает. Схожесть имен – не более чем уловка маркетологов (первоначальное название языка – LiveScript – было изменено на JavaScript в последнюю минуту).

Однако JavaScript и Java действительно хорошо дополняют друг друга, различаясь по своим возможностям. JavaScript может управлять поведением броузера и содержимым страницы, однако не может рисовать графику и работать с сетью. Java не может управлять броузером в целом, но может создавать графику, работать с сетью и позволяет реализовать многопоточность. Клиентский JavaScript может взаимодействовать и управлять встроенными в веб-страницу Java-апплетами и, в этом смысле, реализовать сценарии Java (подробности см. в главе 22).

1.1.2. JavaScript не простой язык

JavaScript позиционируется как язык сценариев, а не язык программирования, при этом подразумевается, что языки сценариев проще и предназначены для непрограммистов. В самом деле, на первый взгляд JavaScript может показаться довольно простым языком, по сложности сравнимым с BASIC. Некоторые возможности JavaScript призваны сделать язык менее строгим и более легким для новичков и неопытных программистов. Непрограммисты могут применять JavaScript для ограниченного круга задач, выполняемых по точным рецептам.

Однако за внешней простотой JavaScript скрывается полноценный язык программирования, столь же сложный, как любой другой, и даже более сложный, чем некоторые. Программисты, пытающиеся решать с помощью JavaScript нетривиальные задачи, часто разочаровываются в процессе разработки из-за того, что недостаточно понимают возможности этого языка.

Данная книга содержит всеобъемлющее описание JavaScript, позволяющее вам стать искушенным знатоком.

1.2. Версии JavaScript

JavaScript развивался много лет, и компания Netscape выпустила несколько версий этого языка. Microsoft выпустила похожие версии языка JavaScript под именем «Jscript». Организация ECMA (*http://www.ecma.ch*) опубликовала три версии ECMA-262, стандартизующие язык JavaScript под неуклюжим названием «ECMAScript».

Версии языка, их ключевые возможности и связь друг с другом описаны в табл. 1.1. В этой книге под названием «JavaScript» я часто понимаю любую реализацию языка, в том числе Microsoft JScript. Для того чтобы сослаться на ECMAScript явно, я употребляю термины «ECMA-262» или «ECMA».

Таблица 1.1. Версии JavaScript

Версия	Описание
JavaScript 1.0	Первоначальная версия языка. Содержала много ошибок и сейчас признана совершенно устаревшей. Реализована в Netscape 2.
JavaScript 1.1	Включает настоящий объект Array; большинство серьезных ошибок устранены. Реализована в Netscape 3.
JavaScript 1.2	Введена инструкция `switch`, регулярные выражения и некоторые другие возможности. В основном совместима с ECMA v1, однако содержит некоторые несоответствия. Реализована в Netscape 4.
JavaScript 1.3	Исправление несовместимостей JavaScript 1.2. Версия совместима с ECMA v1. Реализована в Netscape 4.5.
JavaScript 1.4	Реализована только в серверных продуктах компании Netscape.
JavaScript 1.5	Введена обработка исключений. Совместима с ECMA v3. Реализована в Mozilla и Netscape 6.
JScript 1.0	Практически эквивалентна JavaScript 1.0. Реализована в ранних выпусках IE 3.
JScript 2.0	Практически эквивалентна JavaScript 1.1. Реализована в поздних выпусках IE 3.
JScript 3.0	Практически эквивалентна JavaScript 1.3. Совместима с ECMA v1. Реализована в IE 4.
JScript 4.0	Не реализована ни в одном веб-броузере.
JScript 5.0	Поддерживает обработку исключений. Частично совместима с ECMA v3. Реализована в IE 5.
JScript 5.5	В основном эквивалентна JavaScript 1.5. Полностью совместима с ECMA v3. Реализована в IE 5.5 и IE 6. (IE 6 фактически реализует JScript 5.6, однако версия 5.6 ничем не отличается от 5.5 в том, что касается программистов, использующих клиентский JavaScript.)

Версия	Описание
ECMA v1	Первая стандартная версия языка. Стандартизованы базовые возможности JavaScript 1.1 и добавлены несколько новых возможностей. Не стандартизована инструкция `switch` и поддержка регулярных выражений. Соответствующие реализации – JavaScript 1.3 и Jscript 3.0.
ECMA v2	Промежуточная версия стандарта, включавшая некоторые разъяснения, но не определявшая новых возможностей.
ECMA v3	Стандатизована инструкция `switch`, регулярные выражения и обработка исключений. Соответствующие реализации – JavaScript 1.5 и JScript 5.5.

1.3. Клиентский JavaScript

Когда интерпретатор JavaScript встраивается в веб-броузер, результатом является клиентский JavaScript. Это, безусловно, наиболее распространенный вариант JavaScript, и большинство людей, упоминая JavaScript, обычно подразумевают именно клиентский JavaScript. В этой книге описывается клиентский JavaScript вместе с базовым языком JavaScript, который представляет собой подмножество клиентского JavaScript.

Далее в этой главе мы обсудим клиентский JavaScript и его возможности значительно более подробно. Если же говорить кратко, клиентский JavaScript объединяет способность интерпретатора JavaScript исполнять сценарии с объектной моделью документа (DOM), определяемой веб-броузером. Эти две различные технологии объединяются синергетически, поэтому результат превосходит сумму частей: клиентский JavaScript позволяет распространять исполняемое содержимое по сети и является основой нового поколения динамических HTML-документов (DHTML).

Спецификация ECMA-262 определила стандартную версию базового языка JavaScript, а организация World Wide Web Consortium (W3C) опубликовала спецификацию (или рекомендацию) DOM, стандартизующую возможности, которые броузер должен поддерживать в своей объектной модели. В главах с 17 по 19 содержится намного более подробное обсуждение этого стандарта. Хотя стандарт W3C DOM пока не поддерживается в полном объеме, существующей поддержки достаточно, чтобы веб-разработчики уже могли писать код JavaScript на его основе.

В табл. 1.2 представлена базовая версия языка и возможности DOM, поддерживаемые различными версиями броузеров от Netscape и Microsoft. Обратите внимание, что номера версий Internet Explorer, перечисленные в таблице, относятся к Windows-версиям этого броузера. Возможности версий IE для Macintosh часто отличаются (иногда в значительной степени) от версий с теми же номерами для Windows. Также имейте в виду, что IE позволяет обновлять интерпретатор JScript независимо от самого броузера, поэтому можно

встретить конфигурацию IE, поддерживающую более новую версию языка, чем указанная в таблице.

Таблица 1.2. Возможности клиентского JavaScript для разных броузеров

Броузер	Версия языка	Возможности DOM
Netscape 2	JavaScript 1.0	Работа с формами
Netscape 3	JavaScript 1.1	Переключение изображений
Netscape 4	JavaScript 1.2	DHTML с уровнями
Netscape 4.5	JavaScript 1.3	DHTML с уровнями
Netscape 6 / Mozilla	JavaScript 1.5	Поддержка стандарта W3C DOM в основном; прекращение поддержки уровней
IE 3	JScript 1.0/2.0	Работа с формами
IE 4	JScript 3.0	Переключение изображений; DHTML с `document.all[]`
IE 5	JScript 5.0	DHTML с `document.all[]`
IE 5.5	JScript 5.5	Частичная поддержка стандарта W3C DOM
IE 6	JScript 5.5	Частичная поддержка стандарта W3C DOM; отсутствие поддержки событийной модели W3C DOM

Различия и несовместимость между версиями клиентского JavaScript Netscape и Microsoft значительно превосходят различия между соответствующими реализациями базового языка. Однако довольно много средств клиентского JavaScript в обоих броузерах реализовано одинаково. За неимением лучших названий версии клиентского JavaScript часто обозначаются версиями базового языка, используемого в них. Таким образом, в контексте рассмотрения клиентских сценариев термин «JavaScript 1.2» относится к версии клиентского JavaScript, поддерживаемого в Netscape 4 и IE 4. Если номерами версий базового языка обозначаются клиентские версии JavaScript, подразумевается совместимое подмножество средств, поддерживаемых и Netscape и Internet Explorer. Когда я обсуждаю клиентские возможности броузера, я указываю его название и версию.

Заметьте, что Netscape и IE – не единственные броузеры, поддерживающие клиентский JavaScript. Например, Opera (*http://www.opera.com*) также поддерживает клиентский JavaScript. Однако поскольку Netscape и IE владеют львиной долей рынка броузеров, они и обсуждаются в этой книге. Реализации клиентского JavaScript в других броузерах должны достаточно близко соответствовать реализациям в IE и Netscape.

Кроме того, JavaScript – не единственный язык программирования, который может быть встроен в веб-броузер. Например, Internet Explorer поддерживает язык, известный как VBScript – вариант языка Microsoft Visual Basic, предоставляющий во многом те же возможности, что и JavaScript, но используемый только в броузерах компании Microsoft. Кроме того, в специфи-

кации HTML 4.0 при описании HTML-тега `<script>` в качестве примера встроенного языка сценариев приводится язык программирования Tcl. Среди основных броузеров ни один не поддерживает Tcl, но нет причины, по которой в броузере нельзя было бы реализовать такую поддержку.

В предыдущих изданиях этой книги броузеры Netscape рассматривались более подробно, чем броузеры Microsoft. Дело в том, что JavaScript был создан в Netscape, которая (по крайней мере в течение определенного времени) занимала доминирующую позицию на рынке веб-броузеров. Крен в сторону Netscape с каждым переизданием этой книги уменьшался, и здесь обсуждение в значительной степени сосредоточено на стандартах, таких как ECMAScript и W3C DOM, а не на конкретных броузерах. Тем не менее в материале, оставшемся от ранних изданий, все еще можно заметить некоторое преобладание сведений, относящихся к Netscape.

1.4. JavaScript в иных контекстах

JavaScript – это универсальный язык программирования, и его использование не ограничивается веб-броузерами. Он был разработан для встраивания в любое приложение и обеспечения возможности написания в нем сценариев. Фактически с самого начала веб-серверы Netscape включали интерпретатор JavaScript, что позволяло создавать серверные сценарии на этом языке. Аналогично, Microsoft применяет интерпретатор JScript в своем веб-сервере IIS и в продукте Windows Scripting Host, а не только в Internet Explorer.

И Netscape и Microsoft сделали свои интерпретаторы JavaScript доступными для компаний и программистов, которые хотят встраивать их в свои приложения. Интерпретатор Netscape был выпущен в виде открытого исходного кода и доступен теперь через организацию Mozilla (*http://www.mozilla.org/js/*). Фактически Mozilla предоставляет две различные версии интерпретатора JavaScript. Одна написана на C и называется «SpiderMonkey», а другая – на Java и, что лестно автору этой книги, называется «Rhino» (носорог).

Можно ожидать, что количество приложений, использующих JavaScript в качестве встроенного языка сценариев, будет постоянно увеличиваться.[1] Те, кто пишет сценарии для подобного приложения, найдут полезной первую часть этой книги, описывающую базовый язык. Тогда как материал глав, относящихся к веб-броузерам, видимо, им не понадобится.

---

[1] ActionScript, язык сценариев, доступный в Macromedia Flash 5 и MX, смоделирован в соответствии со стандартом ECMAScript, однако фактически не является JavaScript.

1.5. Клиентский JavaScript: исполняемое содержимое веб-страниц

Веб-броузер, дополненный интерпретатором JavaScript, позволяет распространять по Интернету исполняемое содержимое в виде сценариев JavaScript. В примере 1.1 показана простая JavaScript-программа, или сценарий, встроенный в веб-страницу.

Пример 1.1. Простая программа на JavaScript

```
<html>
<body>
<head><title>Factorials</title></head>
<script language="JavaScript">
document.write("<h2>Table of Factorials</h2>");
for(i = 1, fact = 1; i < 10; i++, fact *= i) {
    document.write(i + "! = " + fact);
    document.write("<br>");
}
</script>
</body>
</html>
```

Будучи загруженным в броузер, поддерживающий JavaScript, этот сценарий выдаст результат, показанный на рис. 1.1.

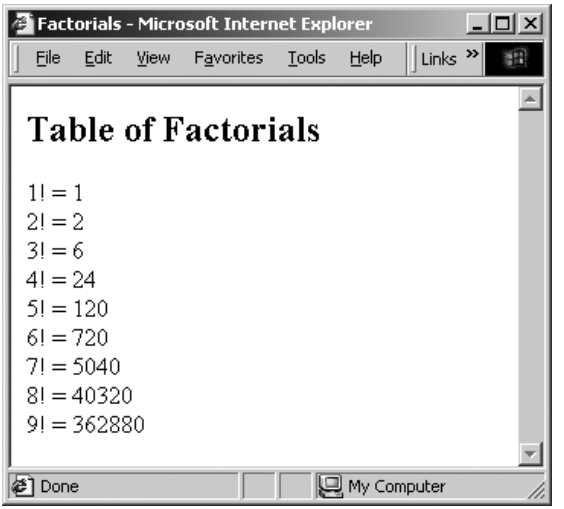

Рис. 1.1. *Веб-страница, сгенерированная JavaScript*

Как видно из этого примера, для встраивания кода JavaScript в HTML-код применяются теги `<script>` и `</script>`. О теге `<script>` мы узнаем больше в главе 12. Главная возможность JavaScript, демонстрируемая данным при-

мером, – это использование метода `document.write()`.[1] Этот метод осуществляет динамический вывод HTML-текста, который анализируется и отображается веб-броузером, и мы еще не раз встретимся с ним в этой книге.

JavaScript обеспечивает возможность управления не только содержимым веб-страниц, но и броузером и содержимым HTML-форм, присутствующих в броузере. Далее в этой главе мы подробнее узнаем об этих возможностях JavaScript, а еще более подробно – в других главах этой книги.

JavaScript может управлять не только содержимым HTML-документов, но и поведением этих документов. Другими словами, JavaScript-программа может реагировать на ввод пользователем значения в текстовое поле ввода или щелчок по изображению в документе. Это достигается путем определения *обработчиков событий* для документа – фрагментов кода JavaScript, выполняемых при возникновении определенного события, например щелчка по кнопке. В примере 1.2 показана простая HTML-форма, включающая обработчик события, который выполняется в ответ на щелчок по кнопке.

Пример 1.2. HTML-форма, для которой определен обработчик события JavaScript

```
<form>
<input type="button"
       value="Click here"
       onclick="alert('You clicked the button');">
</form>
```

На рис. 1.2 отображен результат щелчка по кнопке.

Рис. 1.2. *Отклик JavaScript на событие*

[1] Метод – это объектно-ориентированный термин, обозначающий функцию или процедуру.

Атрибут `onclick` из примера 1.2 изначально представлял собой расширение, введенное в HTML компанией Netscape специально для клиентского JavaScript. Однако теперь этот и другие атрибуты обработчиков событий стандартизованы в HTML версии 4.0. Все обработчики событий JavaScript определяются такими же HTML-атрибутами. Значение атрибута `onclick` – это строка кода JavaScript, выполняемая при щелчке по кнопке. В данном случае обработчик события `onclick` вызывает функцию `alert()`. Как видно на рис. 1.2, функция `alert()` выводит диалоговое окно, отображающее указанное сообщение.

Примеры 1.1 и 1.2 демонстрируют лишь простейшие возможности клиентского JavaScript. Реальная его мощь состоит в том, что сценарии имеют доступ к иерархии объектов, базирующихся на содержимом веб-страницы. Так, клиентские программы на JavaScript могут обращаться к любым изображениям, присутствующим в документе, и управлять ими, устанавливать связь и взаимодействовать с апплетами Java и другими объектами, встроенными в документ HTML. После освоения базового языка JavaScript ключом к его эффективному использованию на веб-страницах становится изучение предоставляемых броузером возможностей DOM.

1.6. Возможности клиентского JavaScript

Возможным применением JavaScript является написание программ, выполняющих произвольные вычисления. Например, можно написать простые сценарии для вычисления чисел Фибоначчи или для поиска простых чисел. Однако в контексте Интернета и веб-броузеров более интересным применением языка может быть программа, вычисляющая налог с продаж в онлайн-заказе на основе информации, введенной пользователем в HTML-форму. Как уже было сказано, настоящая мощь JavaScript состоит в поддержке языком объектов броузера и документа. Чтобы дать вам представления о потенциале JavaScript, в следующих разделах приведен список и описание наиболее важных возможностей клиентского JavaScript и поддерживаемых им объектов.

1.6.1. Управление внешним видом и содержимым документа

Объект Document языка JavaScript, как мы уже видели, позволяет в процессе обработки документа броузером выводить посредством метода `write()` произвольный код HTML. Так, можно включить в документ текущую дату и время или отображать различное содержимое на различных платформах.

С помощью объекта Document можно формировать документ целиком. Свойства объекта Document позволяют указывать цвет фона, текста и гиперссылок в документе. Это обеспечивает возможность формирования динамических и условных HTML-документов, что особенно полезно в многофреймовых документах. В самом деле, в некоторых случаях динамическая

генерация содержимого фрейма позволяет программе на JavaScript целиком заменить традиционные серверные сценарии.

Internet Explorer 4 и Netscape 4 поддерживают собственные методы создания эффектов Dynamic HTML, позволяющие динамически генерировать, перемещать и изменять содержимое документа. IE 4 также поддерживает полноценную модель DOM, предоставляющую JavaScript доступ к каждому отдельному элементу HTML в документе. IE 5.5 и Netscape 6 поддерживают стандарт W3C DOM (или по крайней мере его ключевые части), в котором задается общий, переносимый способ доступа ко всем элементам и тексту в HTML-документе, их размещению и изменению их внешнего вида путем манипуляции с атрибутами таблиц стилей Cascading Style Sheets (CSS). В этих броузерах клиентский JavaScript имеет полную власть над содержимым документа, открывая неограниченные возможности мира сценариев.

1.6.2. Управление броузером

Управлять поведением броузера можно посредством нескольких объектов JavaScript. Объект Window поддерживает методы отображения диалоговых окон с простыми сообщениями и полями для ввода пользовательских данных. В этом объекте также определен метод для создания и открытия (и закрытия) совершенно новых окон броузера, которые могут иметь любой заданный размер и любую комбинацию пользовательских элементов управления. Это позволяет, например, открывать несколько окон и дать пользователю несколько представлений вашего сайта. Новые окна броузера также полезны для временного отображения сгенерированного HTML и, будучи созданными без строки меню и других пользовательских элементов управления, могут служить в качестве диалоговых окон для более сложных сообщений и полей ввода.

JavaScript не определяет методы, позволяющие непосредственно создавать фреймы и манипулировать ими в окне броузера. Однако возможность динамической генерации HTML позволяет программно выводить HTML-теги, создающие любую схему расположения фреймов.

JavaScript также позволяет определять, какие веб-страницы отображаются в броузере. Объект Location позволяет загружать и отображать содержимое любого URL в любом окне или фрейме броузера. Объект History обеспечивает перемещение вперед и назад по истории посещения страниц, имитируя действия кнопок Forward и Back.

Еще один метод объекта Window позволяет JavaScript выводить произвольные сообщения в строку состояния любого окна броузера.

1.6.3. Взаимодействие с HTML-формами

Еще одним важным аспектом клиентского JavaScript является возможность его взаимодействия с HTML-формами. Эта возможность предоставляется объектом Form и объектами элементов формы, которые в ней содержатся:

Button, CheckBox, Hidden, Password, Radio, Reset, Select, Submit, Text и Textarea. Эти объекты элементов позволяют читать и писать значения в составляющие форму элементы ввода. Например, в онлайн-каталоге может присутствовать HTML-форма заполнения заказа, в которой средствами JavaScript осуществляется чтение входных данных формы при вычислении стоимости заказа, налога с продаж и стоимости доставки. Подобные программы JavaScript действительно очень распространены в Интернете. Скоро мы увидим программу, в которой сочетание HTML-формы и JavaScript обеспечивает пользователю возможность вычисления ежемесячных платежей по закладной дома или другой ссуде. В таких приложениях JavaScript имеет явное преимущество перед серверными сценариями. Код JavaScript выполняется на стороне клиента, поэтому содержимое формы не требуется отправлять на сервер для выполнения относительно простых вычислений.

Другое распространенное применение клиентского JavaScript в сочетании с формами – проверка данных формы перед их передачей на сервер. Если все необходимые проверки правильности пользовательского ввода могут быть выполнены клиентским JavaScript, нет необходимости в передаче данных на сервер для нахождения тривиальных ошибок и сообщения о них пользователю. Клиентский JavaScript может выполнять предварительную обработку входных данных, что сократит объем передаваемых на сервер данных. В некоторых случаях клиентский JavaScript может совсем исключить необходимость выполнения сценариев на сервере! (С другой стороны, JavaScript и серверные сценарии прекрасно работают вместе. Например, серверная программа так же, как она динамически создает HTML-код, может «на лету» создавать код JavaScript.)

1.6.4. Взаимодействие с пользователем

Важной чертой JavaScript является возможность определять обработчики событий – произвольные фрагменты кода, исполняемого при возникновении определенных событий. Обычно эти события вызываются пользователем, когда, например, он проводит мышью над гиперссылкой, вводит значение в форму или щелкает по кнопке Submit в форме. Это очень важный механизм, поскольку программирование графических интерфейсов, таких как HTML-формы, изначально требует событийной модели. JavaScript может выполнять любые виды действий в ответ на пользовательские события. Типичным примером может быть отображение специального сообщения в строке состояния, когда пользователь помещает курсор мыши на гиперссылку, или отображение подтверждающего диалогового окна при сохранении данных формы.

1.6.5. Чтение и запись клиентского состояния с помощью cookies

Файлы cookies – это небольшой объем данных состояния, постоянно или временно сохраняемый на стороне клиента. Файлы cookies могут передаваться вместе с веб-страницей сервером клиенту, который сохраняет их на локаль-

ной машине. Когда клиент затем запрашивает ту же или связанную с ней веб-страницу, он передает соответствующие cookies обратно на сервер, где их значения могут использоваться для изменения содержимого страницы, отсылаемой обратно клиенту. Cookies позволяют веб-странице или веб-сайту что-то вспомнить о клиенте – например, что пользователь ранее посещал сайт, зарегистрировался на нем, получил пароль или выразил свои предпочтения о цвете и схеме расположения веб-страниц. Cookies помогают вам предоставить информацию о состоянии, отсутствующую в протоколе HTTP, используемом во Всемирной паутине.

Изначально файлы cookies предназначались исключительно для серверных сценариев, – хотя они сохранялись на стороне клиента, читать их или записывать в них мог только сервер. JavaScript изменил это правило, так как программы JavaScript могут читать и записывать значения cookies и динамически генерировать содержимое документа, базируясь на этих значениях.

1.6.6. Другие возможности

Кроме средств, которые мы уже обсудили, JavaScript имеет много других возможностей, в том числе следующие:

- JavaScript позволяет управлять отображением графики, выводимой с помощью тега `<img>`, создавать анимационные эффекты и переключать изображения.
- JavaScript может взаимодействовать с Java-апплетами и другими встроенными объектами, присутствующими в броузере. Код JavaScript может читать и писать свойства этих апплетов и объектов, а также вызывать любые методы, определенные в них. Эта возможность действительно позволяет JavaScript придавать Java-апплетам свойства сценариев.
- Мы уже говорили, что JavaScript может выполнять произвольные вычисления. В JavaScript имеется тип данных с плавающей точкой, арифметические операторы для работы с ним и полный набор стандартных математических функций для чисел с плавающей точкой.
- Объект Date в JavaScript упрощает процесс вычислений и работы с датами и временем.
- Объект Document поддерживает свойство, содержащее дату последней модификации текущего документа, и с его помощью можно организовать автоматическое отображение временной метки любого документа.
- В JavaScript имеется метод `window.setTimeout()`, позволяющий любому блоку кода JavaScript исполняться на некоторое количество миллисекунд позднее. Это может быть полезно для создания в JavaScript-программе задержек или выполнения повторяющихся действий. В JavaScript 1.2 метод `setTimeout()` дополнен другим полезным методом – `setInterval()`.
- В объекте Navigator (естественно, названном в честь веб-броузера Netscape) имеются переменные, содержащие название и версию запущенного броузера, а также переменные, обозначающие платформу, на которой он

работает. Эти переменные позволяют сценариям настраивать свое поведение в зависимости от броузера или платформы, чтобы сценарий мог воспользоваться преимуществами дополнительных возможностей, поддерживаемых некоторыми версиями, или чтобы обойти ошибки, существующие на некоторых платформах.

- В клиентском JavaScript 1.2 объект Screen предоставляет информацию о размере и глубине цвета монитора, на котором отображается веб-броузер.
- Что касается JavaScript 1.1, метод `scroll()` объекта Window позволяет программам на JavaScript прокручивать окна в размерностях X и Y. В JavaScript 1.2 этот метод дополняется набором других методов, позволяющих перемещать и изменять окна броузера.

1.6.7. Что не умеет JavaScript

Клиентский JavaScript обладает впечатляющим списком возможностей. Заметьте, однако, что они ограничены задачами, связанными с броузерами и документами. Клиентский JavaScript применяется в ограниченном контексте и потому не имеет средств, обязательных для самостоятельных языков программирования:

- JavaScript не имеет каких-либо графических возможностей, за исключением мощного средства динамической генерации HTML-кода (в том числе изображений, таблиц, фреймов, форм, шрифтов и т. д.), отображаемого броузером.
- По соображениям безопасности клиентский JavaScript не допускает чтения или записи файлов. Очевидно, что мало кто захочет, чтобы программа с любого произвольного веб-сайта запустилась на его компьютере и манипулировала файлами по своему усмотрению!
- JavaScript не поддерживает сетевых функций, он лишь может заставить броузер загружать произвольные URL и посылать содержимое HTML-форм по сети серверным сценариям и по электронной почте.

1.7. Безопасность в JavaScript

Во всех тех случаях, когда программы (такие как сценарии JavaScript, программы Visual Basic или макросы Microsoft Word) включаются в совместно используемые документы, в особенности в документы, передаваемые по Интернету или по электронной почте, имеется потенциальная опасность проникновения вирусов и других злонамеренных программ. Разработчики JavaScript не упустили из виду вопросы безопасности и позаботились о том, чтобы не дать программам JavaScript способность выполнять вредные действия. Например, как было описано выше, программы клиентского JavaScript не могут читать локальные файлы или выполнять сетевые операции.

Однако по причине сложности среды веб-броузеров в их ранних версиях все-таки возникали некоторые проблемы безопасности. Например, в Netscape 2

было возможно написание кода JavaScript, который автоматически похищал адрес электронной почты любого посетителя страницы, содержащей этот код, и затем автоматически посылал email-сообщение от его имени, без уведомления или запроса подтверждения посетителя страницы. Эта и несколько других дыр в системе безопасности были исправлены. Хотя нет гарантии, что не будут найдены другие «дыры», большинство осведомленных пользователей спокойно разрешают современным броузерам запускать код JavaScript, находящийся на веб-страницах. В главе 21 вопросы безопасности в клиентском JavaScript обсуждаются подробно.

1.8. Пример: вычисление платежей по ссуде с помощью JavaScript

В примере 1.3 приведен листинг полноценной нетривиальной программы на языке JavaScript. Программа вычисляет месячный платеж по закладной дома или другой ссуде на основании размера ссуды, процентной ставки и периода выплаты. Как видите, программа состоит из HTML-формы, которая сделана интерактивной с помощью кода JavaScript. На рис. 1.3 показано, как HTML-форма выглядит при отображении в веб-броузере. Однако рисунок может только передать статический снимок программы. Добавление кода JavaScript делает ее динамической: как только пользователь изменяет сумму ссуды, процентную ставку или количество платежей, код JavaScript заново вычисляет месячный платеж, общую сумму платежей и общий процент, уплаченный за все время ссуды.

Первая половина примера – это форма HTML, аккуратно отформатированная с помощью HTML-таблицы. Заметьте, что для нескольких элементов формы определены обработчики событий `onchange` или `onclick`. Веб-броузер запускает эти обработчики в тот момент, когда пользователь изменяет входные данные или щелкает по кнопке `Compute`, отображаемой на форме. Заметьте, что во всех этих случаях значением атрибута обработчика события является строка кода JavaScript: `calculate()`. Вызываемый обработчик события исполняет этот код, приводящий к вызову функции `calculate()`.

Функция `calculate()` определена во второй половине примера, внутри тегов `<script>`. Функция читает введенные пользователем данные из формы, выполняет математические действия, требуемые для вычисления платежей по ссуде, и отображает результаты этих действий, используя три нижних элемента формы.

Пример 1.3 прост, но на его внимательное рассмотрение стоит потратить время. Вы не должны понимать весь код JavaScript сейчас, однако изучение этого примера должно дать вам хорошее представление о том, как выглядят программы на JavaScript, как функционируют обработчики событий и как код JavaScript может интегрироваться в HTML-формы. Заметьте, что комментарии в HTML-коде заключаются между тегами `<!--` и `-->`, а в коде JavaScript – в строки, начинающиеся с символов `//`.

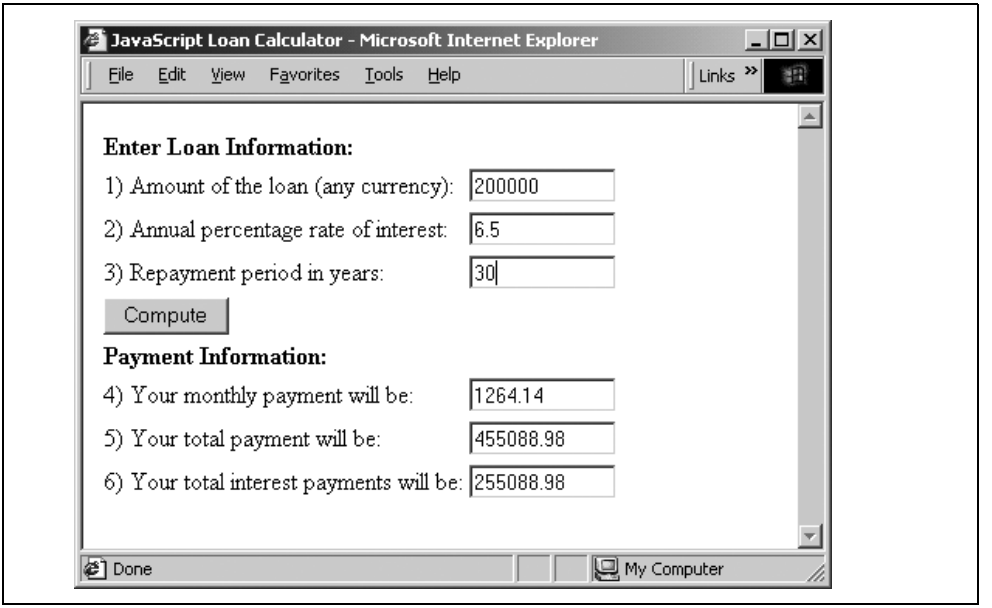

Рис. 1.3. *Калькулятор платежей по ссуде на JavaScript*

Пример 1.3. Вычисление платежей по ссуде с помощью JavaScript

```
<head><title>JavaScript Loan Calculator</title></head>
<body bgcolor="white">
<!--
  Это HTML-форма, дающая пользователю возможность вводить данные и с
  помощью JavaScript показывать ему результат вычислений.
  Элементы формы для улучшения их внешнего вида помещены в таблицу.
  Сама форма имеет имя "loandata", а поля в форме - такие имена, как
  "interest" и "years". Эти имена полей используются в коде JavaScript,
  следующем за формой. Заметьте, что для некоторых элементов формы
  определены обработчики событий "onchange" или "onclick". В них заданы
  строки кода JavaScript, выполняемого при вводе данных или
  щелчке по кнопке.
-->
<form name="loandata">
  <table>
    <tr><td colspan="3"><b>Enter Loan Information:</b></td></tr>
    <tr>
      <td>1)</td>
      <td>Amount of the loan (any currency):</td>
      <td><input type="text" name="principal" size="12"
                 onchange="calculate();"></td>
    </tr>
    <tr>
      <td>2)</td>
```

```
      <td>Annual percentage rate of interest:</td>
      <td><input type="text" name="interest" size="12"
                 onchange="calculate();"></td>
    </tr>
    <tr>
      <td>3)</td>
      <td>Repayment period in years:</td>
      <td><input type="text" name="years" size="12"
                 onchange="calculate();"></td>
    </tr>
    <tr><td colspan="3">
      <input type="button" value="Compute" onclick="calculate();">
    </td></tr>
    <tr><td colspan="3">
      <b>Payment Information:</b>
    </td></tr>
    <tr>
      <td>4)</td>
      <td>Your monthly payment will be:</td>
      <td><input type="text" name="payment" size="12"></td>
    </tr>
    <tr>
      <td>5)</td>
      <td>Your total payment will be:</td>
      <td><input type="text" name="total" size="12"></td>
    </tr>
    <tr>
      <td>6)</td>
      <td>Your total interest payments will be:</td>
      <td><input type="text" name="totalinterest" size="12"></td>
    </tr>
  </table>
</form>

<!--
  Это программа на JavaScript, которая заставляет пример работать.
  Заметьте, что в этом сценарии определяется функция calculate(),
  вызываемая обработчиками событий в форме. Функция ссылается на значения
  в форме, используя имена, определенные в коде, который приведен выше.
-->
<script language="JavaScript">
function calculate() {
    // Получаем пользовательские данные из формы. Предполагаем, что
    // данные являются корректными. Преобразуем процентную ставку из
    // процентов в десятичное значение. Преобразуем период платежа в
    // годах в количество месячных платежей.
    var principal = document.loandata.principal.value;
    var interest = document.loandata.interest.value / 100 / 12;
    var payments = document.loandata.years.value * 12;

    // Теперь вычисляем сумму месячного платежа.
```

```
    var x = Math.pow(1 + interest, payments);
    var monthly = (principal*x*interest)/(x-1);

    // Проверяем, является ли результат конечным числом. Если да, то
    // отображаем результаты.
    if (!isNaN(monthly) &&
        (monthly != Number.POSITIVE_INFINITY) &&
        (monthly != Number.NEGATIVE_INFINITY)) {

        document.loandata.payment.value = round(monthly);
        document.loandata.total.value = round(monthly * payments);
        document.loandata.totalinterest.value =
            round((monthly * payments) - principal);
    }
    // В противном случае пользовательские данные, видимо, не корректны,
    // поэтому не показываем ничего.
    else {
        document.loandata.payment.value = "";
        document.loandata.total.value = "";
        document.loandata.totalinterest.value = "";
    }
}

// Этот простой метод округляет число до двух десятичных знаков.
function round(x) {
  return Math.round(x*100)/100;
}
</script>
</body>
</html>
```

1.9. Структура этой книги

Остальной материал данной книги представлен пятью частями. Часть I, следующая непосредственно за этой главой, содержит описание базового языка JavaScript. Главы 2–6 начинают этот раздел простым, но необходимым материалом – в них приведена базовая информация, необходимая при изучении нового языка программирования:

- Глава 2 «Лексическая структура» описывает базовую структуру языка.
- Глава 3 «Типы данных и значения» описывает типы данных, поддерживаемые JavaScript.
- Глава 4 «Переменные» содержит сведения о переменных, диапазоне их действия и охватывает другие связанные с этим темы.
- Глава 5 «Выражения и операторы» содержит описание выражений JavaScript и всех операторов, поддерживаемых JavaScript. За основу синтаксиса JavaScript взят язык Java, который, в свою очередь, создавался

на основе C и C++, поэтому программисты, имеющие опыт работы с C, C++ и Java могут лишь бегло просмотреть большую часть этой главы.

- Глава 6 «Инструкции» описывает синтаксис и применение каждой из инструкций JavaScript. Опять же, программисты на C, C++ и Java могут бегло просмотреть некоторые разделы, но не всю эту главу.

Следующие пять глав первой части более интересны. В них продолжено рассмотрение ядра языка JavaScript, но описываются те аспекты языка, с которыми не знакомы даже те, кто знает C или Java. Тому, кто действительно хочет понимать JavaScript, эти главы следует изучить тщательно.

- Глава 7 «Функции» описывает порядок определения и вызовов функций в JavaScript и других манипуляций с ними.
- Глава 8 «Объекты» содержит документацию по объектам, наиболее важному типу данных JavaScript. В этой главе обсуждается объектно-ориентированное программирование в JavaScript и объясняется, как в JavaScript можно определять собственные классы объектов.
- Глава 9 «Массивы» описывает создание массивов и работу с ними в JavaScript.
- Глава 10 «Регулярные выражения» описывает применение регулярных выражений в JavaScript для поиска по маске и операций поиска/замены.
- Глава 11 «Прочие вопросы программирования на JavaScript» рассматривает более сложные темы, не затронутые в других местах. При первом чтении эту главу можно пропустить, однако материал, содержащийся в ней, важен для понимания, если вы хотите стать экспертом по JavaScript.

В части II описан клиентский JavaScript. Ее главы документируют объекты веб-броузера, составляющие основу клиентского JavaScript, и содержат подробные примеры работы с ними. Любая представляющая интерес программа JavaScript, работающая в веб-броузере, в значительной степени полагается на возможности, специфичные для клиентской среды.

Вот что вы найдете в части II:

- Глава 12 «JavaScript в веб-броузерах» посвящена интеграции JavaScript в веб-броузеры. В ней обсуждается веб-броузер как программная среда и приведены различные способы интеграции JavaScript в веб-страницы для исполнения на стороне клиента.
- Глава 13 «Окна и фреймы» документирует наиболее существенный и важный объект клиентского JavaScript – объект Window, а также несколько важных объектов, связанных с окном.
- Глава 14 «Объект Document» описывает объект Document и связанные с ним объекты, предоставляющие содержимое HTML-документа для кода JavaScript.
- Глава 15 «Формы и элементы форм» описывает объект Form, представляющий HTML-формы. В ней документируются различные объекты эле-

ментов, присутствующие в HTML-формах, и приведены примеры программирования форм в JavaScript.

- Глава 16 «Сценарии и cookies» иллюстрирует использование файлов cookie в веб-программировании для сохранения состояния.
- Глава 17 «Объектная модель документа» содержит описание базовых частей стандарта W3C DOM и показывает, как программа на JavaScript может обращаться к любым элементам документа HTML.
- Глава 18 «Каскадные таблицы стилей и Dynamic HTML» содержит описание частей стандарта W3C DOM, позволяющих программе на JavaScript управлять стилем, внешним видом и местоположением элементов внутри HTML-документа. В этой главе показано, как создавать многочисленные эффекты DHTML с помощью свойств CSS.
- Глава 19 «События и обработчики событий» рассматривает события JavaScript и обработчики событий, очень важные для всех JavaScript-программ, взаимодействующих с пользователем. В этой главе описываются традиционная событийная модель, событийная модель стандарта W3C DOM и собственная событийная модель Internet Explorer.
- Глава 20 «Приемы обеспечения совместимости» исследует важный вопрос совместимости в программировании на JavaScript и содержит обсуждение приемов написания программ JavaScript, корректно работающих (или изящно сбоящих) на широком разнообразии веб-броузеров.
- Глава 21 «Безопасность в JavaScript» посвящена рассмотрению ограничений, встроенных в клиентский JavaScript из соображений безопасности, и обоснованию их необходимости.
- Глава 22 «Совместное применение Java и JavaScript» содержит обсуждение средств JavaScript, позволяющих организовать управление Java-апплетами. В ней также рассматривается обратная задача – вызов кода JavaScript из Java-апплетов.

Части III, IV и V представляют собой справочники, документирующие объекты, определяемые, соответственно, в базовом языке JavaScript, в традиционном клиентском JavaScript и в новом стандарте W3C DOM.

1.10. Изучение JavaScript

Реальное изучение нового языка программирования невозможно без написания программ. Рекомендую вам при чтении этой книги опробовать возможности JavaScript в процессе их изучения. Вот несколько приемов, призванных облегчить эти эксперименты.

Наиболее очевидный способ изучения JavaScript – это написание простых сценариев. Одно из достоинств клиентского JavaScript состоит в том, что любой, кто имеет веб-броузер и простейший текстовый редактор, уже имеет полноценную среду разработки. Для того чтобы начать писать программы

на JavaScript, нет необходимости в покупке или загрузке специального программного обеспечения. В начале этой главе вы видели пример с вычислением факториалов. Предположим, вы хотите изменить его так, чтобы он выводил вместо факториалов числа Фибоначчи:

```
<script>
document.write("<h2>Таблица чисел Фибоначчи </h2>");
for (i=0, j=1, k=0, fib =0; i<50; i++, fib=j+k, j=k, k=fib){
    document.write("Фибоначчи ("  + i +  ") = " + fib);
    document.write("<br>");
}
</script>
```

Этот код может показаться запутанным (и не волнуйтесь, если вы пока не понимаете его), но для того чтобы поэкспериментировать с подобными короткими программами, достаточно набрать код и запустить в веб-броузере с помощью локального URL `file:`. Заметьте, что для вывода результатов вычислений применяется метод `document.write()`. Это полезный прием при экспериментах с JavaScript. В качестве альтернативы для отображения текстового результата в диалоговом окне можно использовать метод `alert()`:

```
alert("Fibonacci ("  + i +  ") = " + fib);
```

Обратите внимание, что в подобных простых экспериментах с JavaScript можно опускать теги `<html>`, `<head>` и `<body>` в HTML-файле.

Для еще большего упрощения экспериментов с JavaScript можно использовать URL с псевдопротоколом `javascript:` для вычисления значения выражения JavaScript и получения результата. URL JavaScript состоит из признака протокола `javascript:`, за которым следует произвольный код JavaScript (инструкции отделены одна от другой точками с запятой). Загружая такой URL, броузер исполняет код JavaScript. Значение последнего выражения в таком URL преобразуется в строку, и эта строка выводится веб-броузером в качестве нового документа. Например, для того чтобы проверить свое понимание некоторых операторов и инструкций JavaScript, можно набрать следующие URL JavaScript в поле `Location` веб-броузера:

```
javascript:5%2
javascript:x = 3; (x < 5)? "x is less": "x is greater"
javascript:d = new Date(); typeof d;
javascript:for(i=0,j=1,k=0,fib=1; i<10; i++,fib=j+k,k=j,j=fib) alert(fib);
javascript:s=""; for(i in document) s+=i+":"+document[i]+"\n"; alert(s);
```

Не любой ваш код, написанный при изучении JavaScript, будет работать так, как вы ожидаете, и вам захочется его отладить. Базовая методика отладки для JavaScript совпадает с методикой для многих других языков: вставка в код инструкций, которые будут выводить значения нужных переменных так, чтобы можно было понять, что же на самом деле происходит.

Как мы уже видели, иногда для этого можно применить метод `document.write()`. Однако этот метод не годится для обработчиков событий и имеет другие недостатки, так что часто для вывода отладочных сообщений в отдельном диалоговом окне бывает проще использовать функцию `alert()`.

Цикл `for/in` (описанный в главе 6) также полезен в отладке. Например, его можно применять вместе с методом `alert()` для написания функции, отображающей имена и значения всех свойств объекта. Такая функция может быть удобна при изучении языка или при отладке кода.

Удачи вам с JavaScript и счастливой работы!

Часть I. Базовый JavaScript

Данная часть книги включает главы со 2 по 11 и документирует базовый язык JavaScript в том виде, в каком он используется в веб-броузерах, веб-серверах и других встроенных реализациях JavaScript. Этот материал задуман как справочный, и, прочитав главы этой части один раз, вы, возможно, будете неоднократно возвращаться к ним, чтобы освежить в памяти некоторые особенности языка.

- Глава 2 «Лексическая структура»
- Глава 3 «Типы данных и значения»
- Глава 4 «Переменные»
- Глава 5 «Выражения и операторы»
- Глава 6 «Инструкции»
- Глава 7 «Функции»
- Глава 8 «Объекты»
- Глава 9 «Массивы»
- Глава 10 «Регулярные выражения»
- Глава 11 «Прочие вопросы программирования на JavaScript»

2

Лексическая структура

Лексическая структура языка программирования – это набор элементарных правил, определяющих, как пишутся программы на этом языке. Это низкоуровневый синтаксис языка; он задает вид имен переменных, символы, используемые для комментариев, и как одна инструкция отделяется от другой. Эта короткая глава документирует лексическую структуру JavaScript.

2.1. Набор символов

При написании программ на JavaScript используется набор символов Unicode. В отличие от 7-битовой кодировки ASCII, подходящей только для английского языка, и 8-битовой кодировки ISO Latin-1, подходящей только для английского и основных западноевропейских языков, 16-битовая кодировка Unicode обеспечивает представление практически любого письменного языка. Эта возможность важна для интернационализации и особенно важна для программистов, не говорящих на английском языке.

Американские и другие англоговорящие программисты обычно пишут программы с помощью текстового редактора, поддерживающего только кодировки ASCII или Latin-1, и потому не имеют легкого доступа к полноценному набору символов Unicode. Однако никаких трудностей это не порождает, поскольку кодировки ASCII и Latin-1 представляют собой подмножества Unicode, и любая программа JavaScript, написанная с помощью этих наборов символов, абсолютно корректна. Программисты, привыкшие рассматривать символы как 8-разрядные значения, могут быть сбиты с толку, узнав, что JavaScript представляет каждый символ с помощью двух байтов, однако на самом деле для программиста это обстоятельство остается незаметным и может просто игнорироваться.

Стандарт ECMAScript v3 допускает использование символов Unicode в любом месте программы на JavaScript. Однако версии 1 и 2 стандарта допускают применение символов Unicode только в комментариях и строковых литералах, заключенных в кавычки, а все остальные составляющие программы ECMAScript v1 ограничены набором символов ASCII. Версии JavaScript, предшествующие стандарту ECMAScript, обычно вообще не поддерживают Unicode.

2.2. Чувствительность к регистру

JavaScript – это язык, чувствительный к регистру. Это значит, что ключевые слова, переменные, имена функций и любые другие идентификаторы языка должны всегда содержать одинаковые наборы заглавных и строчных букв. Например, ключевое слово `while` должно набираться как while, а не While или WHILE. Аналогично, `online`, `Online`, `OnLine` и `ONLINE` – это имена четырех различных переменных.

Заметим, однако, что HTML не чувствителен к регистру. По причине близкой связи HTML и клиентского JavaScript это различие может привести к путанице. Многие объекты JavaScript и их свойства имеют те же имена, что и теги и атрибуты HTML, которые они обозначают. Если в HTML эти теги и атрибуты могут набираться в любом регистре, то в JavaScript они обычно должны набираться строчными буквами. Например, атрибут обработчика события `onclick` чаще всего задается в HTML как `onClick`, однако в коде JavaScript он должен быть обозначен как `onclick`.

Если базовый JavaScript полностью чувствителен к регистру, то в клиентском JavaScript допускаются исключения из этого правила. Например, в Internet Explorer 3 все клиентские объекты и свойства были не чувствительны к регистру. Однако это приводило к нарушениям совместимости с Netscape, поэтому в Internet Explorer 4 и более поздних версиях клиентские объекты и их свойства стали чувствительными к регистру.

2.3. Символы-разделители и переводы строк

JavaScript игнорирует пробелы, табуляции и переводы строк, присутствующие между лексемами в программе, кроме тех, которые являются частью строковых литералов и регулярных выражений. *Лексема* – это ключевое слово, имя переменной, число, имя функции или какая-либо другая сущность, внутри которых не допускается присутствие пробела или перевода строки. Если в лексему поместить пробел, табуляцию или перевод строки, то получатся две лексемы. Например, `123` – это одна числовая лексема, а `12 3` – это две отдельных лексемы (и, между прочим, такая запись синтаксически ошибочна).

Поскольку допускается наличие в программе пробелов, знаков табуляции и переводов строк (кроме строковых литералов, регулярных выражений и лексем), программист может отформатировать текст четким и непротиворе-

чивым образом, облегчающим чтение и понимание кода. Заметьте, однако, что есть одно небольшое ограничение в расположении переводов строк; оно описано в следующем разделе.

2.4. Необязательные точки с запятой

Простые инструкции JavaScript обычно завершаются точками с запятой (;), как в C, C++ и Java. Точка с запятой служит для отделения инструкций друг от друга. Однако в JavaScript точку с запятой можно не ставить, если каждая инструкция помещается на отдельной строке. Например, следующий код может быть записан без точек с запятой:

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

Однако если обе инструкции расположены в одной строке, то первая точка с запятой должна присутствовать обязательно:

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

Пропуск точек с запятой нельзя признать правильной практикой программирования, и поэтому надо выработать привычку их использовать.

Теоретически JavaScript допускает переводы строк между любыми двумя лексемами, но привычка синтаксического анализатора JavaScript автоматически вставлять точки с запятой за программиста приводит к некоторым исключениям из этого правила. Если в результате разделения строки кода та ее часть, которая предшествует символу перевода, оказывается законченной инструкцией, JavaScript может подумать, что точка с запятой пропущена случайно, и вставить ее, изменив смысл программы. К подобным ситуациям, требующим внимания,– относятся, среди прочих, инструкции `return`, `break` и `continue` (описанные в главе 6). Рассмотрим, например, следующий код:

```
return
true;
```

JavaScript предполагает, что программист имеет в виду следующее:

```
return;
true;
```

Однако он, видимо, хотел написать

```
return true;
```

Вот случай, когда следует быть внимательным, – данный код не вызовет синтаксической ошибки, но приведет к неочевидному сбою. Похожая неприятность возникает, если написать:

```
break
outerloop;
```

JavaScript вставляет точку с запятой после ключевого слова `break`, что вызывает синтаксическую ошибку при попытке интерпретировать следующую строку. По аналогичным причинам постфиксные операторы ++ и -- (см. главу 5) должны располагаться на той же строке, что и выражения, к которым они применяются.

2.5. Комментарии

JavaScript, как и Java, поддерживает комментарии в стиле C++ и в стиле C. Любой текст, присутствующий между символами // и концом строки, рассматривается как комментарий и игнорируется JavaScript. Любой текст между символами /* и */ также рассматривается как комментарий. Эти комментарии в стиле C могут состоять из нескольких строк и не могут быть вложенными. Следующие строки кода представляют собой корректные комментарии JavaScript:

```
// Это однострочный комментарий.
/* Это тоже комментарий */  // а это другой комментарий.
/*
 * Это еще один комментарий.
 * Он располагается в нескольких строках.
 */
```

2.6. Литералы

Литерал – это значение, указанное непосредственно в тексте программы. Ниже приведены примеры литералов:

```
12              // Число двенадцать
1.2             // Число одна целая две десятых
"hello world"   // Строка текста
'Hi'            // Другая строка
true            // Логическое значение
false           // Другое логическое значение
/javascript/gi  // Регулярное выражение (для поиска по маске)
null            // Отсутствие объекта
```

В ECMAScript v3 также поддерживаются выражения, которые могут служить в качестве литералов массивов и объектов. Например:

```
{ x:1, y:2 }    // Инициализатор объекта
[1,2,3,4,5]     // Инициализатор массива
```

Заметьте, что эти литералы массивов и объектов поддерживаются, начиная с JavaScript 1.2, но не были стандартизованы до стандарта ECMAScript v3.

Литералы – важная часть любого языка программирования, поскольку написать программу без них невозможно. Литералы JavaScript описаны в главе 3.

2.7. Идентификаторы

Идентификатор – это имя. В JavaScript они выступают в качестве названий переменных и функций, а также меток некоторых циклов. Правила формирования допустимых идентификаторов совпадают с правилами Java и многих других языков. Первым символом должна быть буква, символ подчеркивания (_) или знак доллара ($).[1] Последующие символы могут быть любой буквой, цифрой, символом подчеркивания или знаком доллара. (Цифра не может быть первым символом, так как тогда интерпретатору труднее отличать идентификаторы от чисел.) Вот примеры допустимых идентификаторов:

```
i
my_variable_name
v13
_dummy
$str
```

В ECMAScript v3 идентификаторы могут содержать буквы и цифры из полного набора Unicode. До этой версии стандарта идентификаторы JavaScript были ограничены набором ASCII. ECMAScript v3 также допускает наличие в идентификаторах escape-последовательностей – символов `\u`, за которыми расположены 4 шестнадцатеричные цифры, обозначающие 16-разрядный код символа. Например, идентификатор π можно записать как `\u03c0`. Этот синтаксис неудобен, но обеспечивает возможность транслитерации программ на JavaScript с символами Unicode в форму, допускающую работу с ними в текстовых редакторах и других средствах, не поддерживающих полный набор Unicode.

И наконец, идентификаторы не могут совпадать ни с одним из ключевых слов, предназначенных в JavaScript для других целей (см. следующий раздел).

2.8. Зарезервированные слова

В JavaScript имеется несколько зарезервированных слов. Они не могут быть идентификаторами (именами переменных, функций и меток циклов) в программах JavaScript. В табл. 2.2.1 перечислены ключевые слова, стандартизованные в ECMAScript v3. Для интерпретатора JavaScript они имеют специальное значение, так как являются частью синтаксиса языка.

Таблица 2.1. Зарезервированные ключевые слова JavaScript

break	do	if	switch	typeof
case	else	in	this	var
catch	false	instanceof	throw	void
continue	finally	new	true	while
default	for	null	try	with
delete	function	return		

[1] Знаки $ не допускаются в идентификаторах для версий, более ранних, чем JavaScript 1.1. Они предназначены только для средств генерации кода.

В табл. 2.2 перечислены другие ключевые слова. В настоящее время они в JavaScript не используются, но зарезервированы ECMAScript v3 в качестве возможных будущих расширений языка.

Таблица 2.2. Слова, зарезервированные для расширений ECMA

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

Кроме нескольких только что перечисленных формально зарезервированных слов, текущие проекты стандарта ECMAScript v4 рассматривают применение ключевых слов `as`, `is`, `namespace` и `use`. Текущие интерпретаторы JavaScript не мешают использовать эти четыре слова в качестве идентификаторов, однако все равно следует этого избегать.

Таблица 2.3. Другие идентификаторы, которых стоит избегать

arguments	encodeURI	Infinity	Object	String
Array	Error	isFinite	parseFloat	SyntaxError
Boolean	escape	isNaN	parseInt	TypeError
Date	eval	Math	RangeError	undefined
decodeURI	EvalError	NaN	ReferenceError	unescape
decodeURIComponent	Function	Number	RegExp	URIError

Не следует также употреблять в качестве идентификаторов имена глобальных переменных и функций, предопределенных в JavaScript. Если создать переменные или функции с этими именами, то будет либо получено сообщение об ошибке (если свойство доступно только для чтения), либо переопределена существующая переменная или функция, – избегайте этого, если только вы не знаете точно, что делаете. В табл. 2.3 перечислены глобальные переменные и функции, определенные в стандарте ECMAScript v3. Конкретные реализации могут определять другие глобальные свойства, и каждая конкретная среда JavaScript (клиентская, серверная и т. д.) будет иметь собственный расширенный список глобальных свойств.[1]

[1] В описании объекта Window в «Справочнике по клиентскому JavaScript» (часть IV) приведен список глобальных переменных и функций, определенных в клиентском JavaScript.

3

Типы данных и значения

Компьютерные программы работают, манипулируя *значениями* (*values*), такими как число 3,14 или текст «Hello World». Типы значений, которые могут быть представлены и обработаны в языке программирования, известны как *типы данных* (*data types*), и одной из наиболее фундаментальных характеристик языка программирования является поддерживаемый им набор типов данных. JavaScript позволяет работать с тремя базовыми типами данных: числами, строками текста (или просто «строками») и значениями логической истинности (или просто «логическими»). В JavaScript также определяются два тривиальных типа данных, null и undefined, каждый из которых определяет только одно значение.

В дополнение к этим базовым типам данных JavaScript поддерживает составной тип данных, известный как объект. Объект (то есть член объектного типа данных) представляет собой коллекцию значений (либо простейших, таких как числа и строки, либо сложных, например других объектов). Объекты в JavaScript имеют двойственную природу: объект может быть представлен как неупорядоченная коллекция именованных значений или как упорядоченная коллекция пронумерованных значений. В последнем случае объект называется *массивом* (*array*). Хотя в JavaScript объекты и массивы в основе являются одним типом данных, они ведут себя совершенно по-разному, и в этой книге будут рассматриваться как отдельные типы.

В JavaScript определен еще один специальный тип объекта, известный как *функция* (*function*). Функция – это объект, с которым связан исполняемый код. Функция может *вызываться* (*invoked*) для выполнения определенной операции. Подобно массивам, функции ведут себя не так, как другие виды объектов, и в JavaScript определен специальный синтаксис для работы с ними. Поэтому мы будем рассматривать функции независимо от объектов и массивов.

Кроме функций и массивов в базовом JavaScript определено еще несколько специальных видов объектов. Эти объекты представляют собой не новые типы данных, а новые *классы* (*classes*) объектов. Класс Date определяет объекты, представляющие даты, класс RegExp – объекты, представляющие регулярные выражения (мощное средство поиска по маске, описываемое в главе 10), и класс Error – объекты, представляющие синтаксические ошибки и ошибки времени выполнения, которые могут возникать в программе на JavaScript.

В оставшейся части этой главы подробно описан каждый из базовых типов данных. В ней также приведены начальные сведения об объектах, массивах и функциях, которые более подробно рассмотрены в главах 7, 8 и 9. И наконец, в ней приведен обзор классов Date, RegExp и Error, подробно документируемых в справочном разделе, посвященном базовому языку.

3.1. Числа

Числа – это основной тип данных, не требующий особых пояснений. JavaScript отличается от таких языков программирования, как C и Java, тем, что не делает различия между целыми значениями и значениями с плавающей точкой. Все числа в JavaScript представляются как значения с плавающей точкой. JavaScript представляет числа с помощью 64-разрядного вещественного формата, определяемого стандартом IEEE 754.[1] Этот формат способен представлять числа от $\pm 1{,}7976931348623157 \times 10^{308}$ до $\pm 5 \times 10^{-324}$.

Число, находящееся непосредственно в коде программы на JavaScript, называется числовым литералом. JavaScript поддерживает числовые литералы нескольких форматов, описанных в последующих разделах. Заметьте, что любому числовому литералу может предшествовать знак «минус» (–), делающий числа отрицательными. Однако фактически «–» представляет собой унарный оператор смены знака (см. главу 5), не являющийся частью синтаксиса числовых литералов.

3.1.1. Целые литералы

В JavaScript целые десятичные числа записываются как последовательность цифр. Например:

```
0
3
10000000
```

[1] Этот формат должен быть знаком Java-программистам как формат типа double. Это также формат double в почти всех современных реализациях C и C++.

Числовой формат JavaScript позволяет точно представлять все целые числа в диапазоне от –9007199254740992 (-2^{53}) до 9007199254740992 (2^{53}), включительно. Для целых значений вне этого диапазона может теряться точность в младших разрядах. Заметьте, однако, что некоторые целые операции в JavaScript (в особенности битовые операторы, описанные в главе 5) выполняются с 32-разрядными целыми, принимающими значения от –2147483648 (-2^{31}) до 2147483647 ($2^{31}-1$).

3.1.2. Шестнадцатеричные и восьмеричные литералы

Кроме десятичных целых литералов JavaScript распознает шестнадцатеричные значения (по основанию 16). Шестнадцатеричные литералы начинаются с последовательности символов «0x» или «0X», за которой следует строка шестнадцатеричных цифр. Шестнадцатеричная цифра – это одна из цифр от 0 до 9 или буква от «a» (или «A») до «f» (или «F»), представляющих значения от 10 до 15. Примеры шестнадцатеричных целых литералов:

```
0xff  // 15*16 + 15 = 255 (по основанию 10)
0xCAFE911
```

Хотя стандарт ECMAScript не поддерживает это, некоторые реализации JavaScript допускают представление целых литералов в восьмеричном формате (по основанию 8). Восьмеричный литерал начинается с цифры 0, за которой следуют цифры, каждая из которых может быть от 0 до 7. Например:

```
0377  // 3*64 + 7*8 + 7 = 255 (по основанию 10)
```

Поскольку некоторые реализации поддерживают восьмеричные литералы, а некоторые нет, никогда не следует писать целый литерал с ведущим нулем – нельзя сказать наверняка, как он будет интерпретирован данной реализацией – как восьмеричное число или как десятичное.

3.1.3. Литералы с плавающей точкой

Литералы с плавающей точкой должны иметь десятичную точку; в них используется традиционный синтаксис вещественных чисел. Вещественное значение представлено как целая часть числа, за которой следуют десятичная точка и дробная часть числа.

Литералы с плавающей точкой могут также представляться в экспоненциальной нотации: вещественное число, за которым следует буква «e» (или «E»), а затем необязательный знак плюс или минус и целая экспонента. Эта нотация обозначает вещественное число, умноженное на 10 в степени, определяемой значением экспоненты.

Более лаконичное определение синтаксиса таково:

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

Например:

```
3.14
2345.789
.333333333333333333
6.02e23        // 6.02 X 10^23
1.4738223E-32  // 1.4738223 X 10^-32
```

Заметьте, что существует бесконечно много вещественных чисел, но формат с плавающей точкой в JavaScript позволяет точно выразить лишь ограниченное их количество (точнее, 18437736874454810627). Это значит, что при работе с вещественными числами в JavaScript представление числа часто будет округлением реального числа. Точность округления, как правило, достаточна, и на практике редко приводит к ошибкам.

3.1.4. Работа с числами

Для работы с числами в программах на JavaScript применяются поддерживаемые языком арифметические операторы. Это операторы сложения (+), вычитания (–), умножения (*) и деления (/). Подробное описание этих и других арифметических операторов приведено в главе 5.

Кроме перечисленных основных арифметических операторов, JavaScript поддерживает выполнение более сложных математических операций с помощью большого количества математических функций, относящихся к базовой части языка. Для удобства эти функции хранятся в виде свойств одного объекта Math, и для доступа к ним всегда используется литеральное имя `Math`. Например, вот как вычисляется синус числового значения переменной `x`:

```
sine_of_x = Math.sin(x);
```

А так вычисляется квадратный корень числового выражения:

```
hypot = Math.sqrt(x*x + y*y);
```

Подробные сведения обо всех математических функциях, поддерживаемых JavaScript, приведены в описании объекта Math и соответствующих листингах справочного раздела, посвященного базовому языку.

Имеется еще один интересный метод, применяемый для работы с числами. Метод `toString()` преобразует целое число в строку, используя основание системы счисления, указанное в качестве его аргумента (оно должно быть в диапазоне от 2 до 36). Вот пример преобразования числа в двоичное представление с помощью метода `toString()`:

```
var x = 33;
var y = x.toString(2);  // y равно "100001"
```

Вызывая метод `toString()` для числового литерала, можно заключить последний в круглые скобки, чтобы предотвратить интерпретацию точки как десятичного разделителя:

```
var y = (257).toString(0x10);  // y равно "101"
```

3.1.5. Специальные числовые значения

В JavaScript определено несколько специальных числовых значений. Когда вещественное число превышает самое большое представимое конечное значение, результату присваивается специальное значение бесконечности, которое в JavaScript обозначается как `Infinity`. А когда отрицательное число становится меньше наименьшего представимого отрицательного числа, результатом является минус бесконечность, обозначаемая как `–Infinity`.

Еще одно специальное числовое значение возвращается JavaScript, когда математическая операция (например, деление нуля на ноль) приводит к неопределенному результату или ошибке. В этом случае результатом является специальное значение «не число», обозначаемое как `NaN`. «Не число» ведет себя необычно: оно не равно ни одному другому числу, в том числе и самому себе! По данной причине для проверки на это значение имеется специальная функция `isNaN()`. Похожая функция, `isFinite()`, позволяет проверить число на равенство NaN или положительной/отрицательной бесконечности.

В табл. 3.1 приведено несколько констант, определенных в JavaScript для обозначения специальных числовых значений.

Таблица 3.1. Специальные числовые константы

Константа	Значение
`Infinity`	Специальное значение, обозначающее бесконечность
`NaN`	Специальное значение «не число»
`Number.MAX_VALUE`	Максимальное представимое значение
`Number.MIN_VALUE`	Наименьшее (ближайшее к нулю) представимое число
`Number.NaN`	Специальное значение «не число»
`Number.POSITIVE_INFINITY`	Специальное значение, обозначающее плюс бесконечность
`Number.NEGATIVE_INFINITY`	Специальное значение, обозначающее минус бесконечность

Константы `Infinity` и `NaN` определяются в ECMAScript v1 и не были реализованы вплоть до JavaScript 1.3. Однако различные константы `Number` реализованы начиная с JavaScript 1.1.

3.2. Строки

Строка представляет собой последовательность букв, цифр, знаков пунктуации и прочих символов Unicode и является типом данных JavaScript для представления текста. Как вы скоро увидите, строковые литералы можно использовать в своих программах, заключая их в согласованные пары одинарных или двойных кавычек. Обратите внимание, что в JavaScript нет символьного типа данных, такого как `char` в C, C++ и Java. Одиночный символ представлен строкой, имеющей длину 1.

3.2.1. Строковые литералы

Строковый литерал – это последовательность из нуля или более символов Unicode, заключенная в одинарные или двойные кавычки (' или "). Символы двойных кавычек могут содержаться в строках, ограниченных символами одинарных кавычек, а символы одинарных кавычек – в строках, ограниченных двойными кавычками. Строковые литералы должны записываться в одной строке программы, и не могут разбиваться на две строки. Для того чтобы включить в строковый литерал символ перевода строки, используйте последовательность символов \n, описание которой приведено в следующем разделе. Примеры строковых литералов:

```
""  // Это пустая строка: в ней ноль символов
'testing'
"3.14"
'name="myform"'
"Вы предпочитаете книги издательства O'Reilly, не правда ли?"
"В этом строковом литерале\nдве строки"
"π - это отношение длины окружности круга к его диаметру"
```

Как иллюстрирует последний пример строки, стандарт ECMAScript v1 допускает символы Unicode в строковых литералах. Однако в реализациях, более ранних, чем JavaScript 1.3, в строках обычно поддерживаются только символы ASCII или Latin-1. Как мы увидим в следующем разделе, символы Unicode также можно включать в строковые литералы с помощью специальных управляющих последовательностей. Это может быть полезно, если в текстовом редакторе отсутствует полноценная поддержка Unicode.

Заметьте, что, ограничивая строку одинарными кавычками, необходимо проявлять осторожность в обращении с апострофами, употребляемыми в английском языке для обозначения притяжательного падежа и в сокращениях, как в словах «can't» и «O'Reilly's». Поскольку апостроф и одиночная кавычка – это одно и то же, необходимо при помощи обратной косой черты (\) экранировать апострофы, расположенные внутри одиночных кавычек (см. следующий раздел).

Программы на клиентском JavaScript часто содержат строки кода HTML, а код HTML часто содержит строки JavaScript. Как и в JavaScript, в HTML для ограничения строк применяются либо одинарные, либо двойные кавычки. Поэтому при объединении JavaScript и HTML есть смысл придерживаться одного «стиля» кавычек для JavaScript, а другого – для HTML. В следующем примере строка «Спасибо» в выражении JavaScript заключена в одинарные кавычки, а само выражение, в свою очередь, заключено в двойные кавычки как значение атрибута обработчика событий HTML:

```
<a href="" onclick="alert('Спасибо')">Нажми меня</a>
```

3.2.2. Управляющие последовательности в строковых литералах

Символ обратной косой черты (\) имеет специальное предназначение в строках JavaScript. Вместе с символами, следующими за ним, он обозначает символ, не представимый внутри строки другими способами. Например, `\n` – это *управляющая последовательность* (*escape sequence*), обозначающая символ перевода строки.[1]

Другой пример, упомянутый в предыдущей главе, – это последовательность `\'`, обозначающая символ одинарной кавычки. Эта управляющая последовательность полезна, когда требуется включить одинарную кавычку в качестве апострофа в строковый литерал, заключенный в одинарные кавычки. Теперь становится понятно, почему мы называем эти последовательности управляющими – здесь обратная косая черта позволяет управлять интерпретацией символа одинарной кавычки. Вместо того чтобы отмечать ею конец строки, мы используем ее как апостроф:

```
'You\'re right, it can\'t be a quote'
```

В табл. 3.2 перечислены управляющие последовательности и символы, обозначаемые ими. Две управляющие последовательности являются обобщенными; они могут применяться для представления любого символа путем указания кода символа Latin-1 или Unicode в виде шестнадцатеричного числа. Например, последовательность `\xA9` обозначает символ копирайта, который в кодировке Latin-1 имеет шестнадцатеричный код A9. Аналогично, управляющая последовательность, начинающаяся с `\u`, обозначает произвольный символ Unicode, заданный четырьмя шестнадцатеричными цифрами. Например, `\u03c0` обозначает символ π. Заметьте, что управляющие последовательности для символов Unicode требуются по стандарту ECMAScript v1, но обычно не поддерживаются в реализациях, вышедших ранее, чем JavaScript 1.3. Некоторые реализации JavaScript также допускают задание символа Latin-1 тремя восьмеричными символами, указанными после обратной косой черты, но такие управляющие последовательности не поддерживаются в стандарте ECMAScript v3 и не должны больше использоваться.

Таблица 3.2. Управляющие последовательности JavaScript

Последовательность	Обозначаемый символ
`\0`	Символ NUL (`\u0000`)
`\b`	«Забой» (`\u0008`)
`\t`	Горизонтальная табуляция (`\u0009`)
`\n`	Перевод строки (`\u000A`)
`\v`	Вертикальная табуляция (`\u000B`)

[1] Тем, кто программирует на C, C++ и Java, эта и другие управляющие последовательности JavaScript уже знакомы.

Таблица 3.2 (продолжение)

Последовательность	Обозначаемый символ
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\"	Двойная кавычка (\u0022)
\'	Одинарная кавычка (\u0027)
\\	Обратная косая черта (\u005C)
\x*XX*	Символ Latin-1, заданный двумя шестнадцатеричными цифрами *XX*
\u*XXXX*	Символ Unicode, обозначенный четырьмя шестнадцатеричными цифрами *XXXX*
XXX	Символ Latin-1, обозначенный восьмеричными цифрами *XXX*, между 1 и 377. Не поддерживается ECMAScript v3; не используйте эту последовательность

И наконец, заметьте, что символ обратной косой черты не может предшествовать символу перевода строки для продолжения строки (или другой лексемы JavaScript) на следующей строке, или включения буквального перевода строки в строковый литерал. Если символ «\» предшествует любому символу, отличному от приведенных в табл. 3.2, косая черта просто игнорируется (хотя будущие версии могут, конечно, определять новые управляющие последовательности). Например, \# – это то же самое, что и #.

3.2.3. Работа со строками

Одной из встроенных возможностей JavaScript является способность *конкатенировать* строки. Если оператор + применяется к числам, они складываются. А если к строкам, то они объединяются, при этом вторая строка добавляется в конец первой. Например:

```
msg = "Hello, " + "world";   // Получается строка "Hello, world"
greeting = "Добро пожаловать на мою домашнюю страницу," + " " + name;
```

Для определения длины строки – количества содержащихся в ней символов – применяется свойство length. Так, если переменная s содержит строку, то длину последней можно получить следующим образом:

```
s.length
```

Для работы со строками существует несколько методов. Так можно получить последний символ в строке s:

```
last_char = s.charAt(s.length - 1)
```

Так – извлечь второй, третий и четвертый символы из строки s:

```
sub = s.substring(1,4);
```

А позицию первого символа «а» в строке s можно получить так:

```
i = s.indexOf('a');
```

Есть и еще ряд методов, которые можно применять для манипуляций со строками. Полностью эти методы документированы в описании объекта String и последующих листингах справочного раздела, посвященного базовому языку.

Из предыдущих примеров можно понять, что строки JavaScript (и, как мы увидим позднее, массивы JavaScript) индексируются начиная с 0. Другими словами, номер первого символа строки равен нулю. Программистам, работавшим с C, C++ и Java, должно быть удобно это соглашение, однако программистам, привыкшим к языкам, в которых нумерация строк и массивов начинается с единицы, придется какое-то время привыкать к этому.

В некоторых реализациях JavaScript отдельные символы могут извлекаться из строк (но не записываться в строки) с помощью нотации массива, так что вышеприведенный вызов метода `charAt()` может быть также записан следующим образом:

```
last_char = s[s.length - 1];
```

Заметьте, однако, что этот синтаксис не стандартизован в ECMAScript v3, не является переносимым и его следует избегать.

Когда мы будем обсуждать объектный тип данных, вы увидите, что свойства и методы объектов используются так же, как в предыдущих примерах свойства и методы строк. Это не значит, что строки – это тип объектов. На самом деле строки – это отдельный тип данных JavaScript. Для доступа к их свойствам и методам используется объектный синтаксис, но сами они объектами не являются. Почему это так, мы узнаем в конце данной главы.

3.3. Логические значения

Числовые и строковые типы данных имеют большое или бесконечное количество возможных значений. Логический тип данных, напротив, имеет только два допустимых логических значения, представленных литералами `true` и `false`. Логическое значение говорит об истинности чего-то, то есть о том, является это что-то истиной или нет.

Логические значения обычно представляют собой результат сравнений, выполняемых в программах JavaScript. Например:

```
a == 4
```

Этот код проверяет, равно ли значение переменной `a` числу `4`. Если да, то результатом этого сравнения будет логическое значение `true`. Если переменная `a` не равна `4`, то результатом сравнения будет `false`.

Логические значения обычно используются в управляющих конструкциях JavaScript. Например, инструкция `if/else` в JavaScript выполняет одно действие, если логическое значение равно `true`, и другое действие, если значение равно `false`. Обычно сравнение, создающее логическое значение, непо-

средственно объединяется с инструкцией, в которой оно используется. Результат выглядит следующим образом:

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

Код проверяет, равна ли переменная a числу 4. Если да, он добавляет 1 к b; в противном случае добавляет 1 к a.

Вместо того чтобы интерпретировать два возможных логических значения как true и false, иногда удобно рассматривать их как «включено» (true) и «выключено» (false) или «да» (true) и «нет» (false). Иногда их даже полезно рассматривать как 1 (true) и 0 (false). (Фактически JavaScript делает именно так, и, когда это необходимо, преобразует true и false в 1 и 0.) [1]

3.4. Функции

Функция – это фрагмент исполняемого кода, который определен в программе на JavaScript или заранее предопределен в реализации JavaScript. Хотя функция определяется только один раз, программа на JavaScript может исполнять или вызывать ее сколько угодно. Функции могут передаваться аргументы, или параметры, задающие значение, или значения, для которых она должна выполнять вычисления, а также функция может возвращать значение, представляющее собой результат этих вычислений. Реализации JavaScript предоставляют много предопределенных функций, таких как Math.sin(), вычисляющая синус угла.

Программы на JavaScript могут также определять свои собственные функции, содержащие, например, такой код:

```
function square(x)  // Функция называется square. Она принимает один аргумент, x.
{                   // Здесь начинается тело функции.
  return x*x;       // Функция возводит свой аргумент в квадрат и возвращает
                    // полученное значение.
}                   // Здесь функция заканчивается.
```

[1] Тем, кто программировал на C, следует обратить внимание, что в JavaScript имеется отдельный логический тип данных, в отличие от C, в котором для имитации логических значений служат целые числа. Java-программистам следует иметь в виду, что хотя в JavaScript имеется логический тип, он не настолько чист, как тип данных boolean в Java, – в JavaScript логические значения легко преобразуются в другие типы данных и обратно, и потому на практике в том, что касается работы с логическими значениями, JavaScript больше напоминает C, чем Java.

Определив функцию, можно вызывать ее, указав имя, за которым следует заключенный в скобки список необязательных аргументов, разделенных запятыми. Следующие строки представляют собой вызовы функций:

```
y = Math.sin(x);
y = square(x);
d = compute_distance(x1, y1, z1, x2, y2, z2);
move();
```

Важной чертой JavaScript является то, что функции представляют собой значения, которыми можно манипулировать в коде JavaScript. Во многих языках, в том числе в Java, функции – это всего лишь синтаксические элементы языка: их можно определять и вызывать, но это не тип данных. То обстоятельство, что функции в JavaScript представляют собой настоящие значения, придает языку большую гибкость. Это означает, что функции могут храниться в переменных, массивах и объектах, а также что функции могут передаваться в качестве аргументов другим функциям. Очень часто это бывает полезным. Более подробно об определении и вызове функций, а также об использовании их в качестве значений рассказано в главе 7.

Поскольку функции представляют собой значения – такие же, как числа и строки, они могут присваиваться свойствам объектов. Когда функция присваивается свойству объекта (объектный тип данных и свойства объекта описаны в разделе 3.5 «Объекты»), она часто называется *методом* этого объекта. Методы – важная часть объектно-ориентированного программирования. Им посвящена глава 8.

3.4.1. Функциональные литералы

В предыдущем разделе мы видели определение функции `square()`. С помощью синтаксиса, описанного там, определено большинство функций в большинстве программ на JavaScript. Однако стандарт ECMAScript v3 предоставляет синтаксис (реализованный в JavaScript 1.2 и более поздних версиях) для определения функциональных литералов. Функциональный литерал задается с помощью ключевого слова `function`, затем следует необязательное имя функции, затем – список аргументов функции, заключенный в круглые скобки, и тело функции в фигурных скобках. Другими словами, функциональный литерал выглядит так же, как определение функции, правда, у него может не быть имени. Самое большое различие состоит в том, что функциональные литералы могут присутствовать в других выражениях JavaScript. То есть функцию `square()` можно задать не в виде определения

```
function square(x) { return x*x; }
```

а с помощью функционального литерала:

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

Функции, определенные таким образом, иногда называют *лямбда-функциями*. Это дань уважения языку программирования LISP, который одним из первых допускал вставку неименованных функций в виде литералов внутрь программы. Хотя в данный момент польза от функциональных литералов может быть неочевидной, позднее, в сложных сценариях мы увидим, что они бывают довольно удобными и полезными.

Имеется еще один способ определения функции: можно передать список аргументов и тело функции в виде строк в конструктор `Function()`. Например:

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

Такое определение функций используется редко. Обычно неудобно задавать тело функции в виде строки, и во многих реализациях JavaScript функции, определенные подобным образом, менее эффективны, чем функции, определенные любым из двух других способов.

3.5. Объекты

Объект – это коллекция именованных значений. Эти именованные значения обычно называются *свойствами* (*properties*) объекта. (Иногда они называются полями объекта, но употребление этого термина может сбить с толку.) Для того чтобы сослаться на свойство объекта, надо указать объект, затем точку, а затем имя свойства. Например, если объект под названием `image` имеет свойства `width` и `height`, мы можем сослаться на эти свойства следующим образом:

```
image.width
image.height
```

Свойства объектов во многом похожи на переменные JavaScript – они могут содержать любой тип данных, включая массивы, функции и другие объекты. Поэтому можно увидеть вот такой код JavaScript:

```
document.myform.button
```

Этот код ссылается на свойство `button` объекта, который, в свой очередь, хранится в свойстве `myform` объекта с именем `document`.

Как упоминалось раньше, функция, хранящаяся в свойстве объекта, часто называется методом, а имя свойства становится именем метода. При вызове метода объекта сначала используется оператор «точка» для извлечения функции, а затем `()` для вызова этой функции. Например, метод `write()` объекта Document можно вызвать так:

```
document.write("это проверка");
```

Объекты в JavaScript могут выступать в качестве ассоциативных массивов, то есть могут ассоциировать произвольные значения с произвольными строками. При такой работе с объектом обычно требуется другой синтаксис для

доступа к его свойствам: строка, содержащая имя требуемого свойства, заключается в квадратные скобки. Тогда к свойствам объекта image, упомянутого выше, можно обратиться посредством следующего кода:

```
image["width"]
image["height"]
```

Ассоциативные массивы – это мощный тип данных; они полезны при реализации ряда технологий программирования. Об объектах, их традиционном применении и применении в качестве ассоциативных массивов рассказано в главе 8.

3.5.1. Создание объектов

Как мы увидим в главе 8, объекты создаются путем вызова специальных функций-*конструкторов*. Все следующие строки создают новые объекты:

```
var o = new Object();
var now = new Date();
var pattern = new RegExp("\\sjava\\s", "i");
```

Создав собственный объект, можно использовать и устанавливать его свойства как угодно:

```
var point = new Object();
point.x = 2.3;
point.y = -1.2;
```

3.5.2. Объектные литералы

В ECMAScript v3 определяется (а в JavaScript 1.2 реализуется) синтаксис объектных литералов, позволяющий создавать объекты и указывать их свойства. Объектный литерал (также называемый инициализатором объекта) представляет собой список разделенных запятыми пар «свойство-значение», заключенный в фигурные скобки. Внутри пар роль разделителя играет двоеточие. Таким образом, объект point из предыдущего примера также может быть создан и инициализирован следующей строкой:

```
var point = { x:2.3, y:-1.2 };
```

Объектные литералы также могут быть вложенными. Например:

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

Кроме того, значения свойств в объектных литералах не обязательно должны быть константами – они могут быть произвольными выражениями:

```
var square = { upperLeft:  { x:point.x, y:point.y },
               lowerRight: { x:(point.x + side), y:(point.y+side) }};
```

3.6. Массивы

Массив (*array*), как и объект, представляет собой коллекцию значений. Если каждое значение, содержащееся в объекте, имеет имя, то в массиве каждое значение имеет номер, или *индекс*. В JavaScript можно извлекать значения из массива, указав после имени массива индекс, заключенный в квадратные скобки. Например, если массив имеет имя `a`, а `i` – это неотрицательное целое число, то `a[i]` – это элемент массива. Индексы массива начинаются с нуля, то есть `a[2]` ссылается на *третий* элемент массива `a`.

Массивы могут содержать любой тип данных JavaScript, в том числе ссылки на другие массивы или на объекты или функции. Например:

```
document.images[1].width
```

Этот код ссылается на свойство `width` объекта, хранящегося во втором элементе массива, хранящегося в свойстве `images` объекта `document`.

Заметьте, что массивы, описанные здесь, отличаются от ассоциативных массивов (см. раздел 3.5 «Объекты»). Настоящие массивы, обсуждаемые здесь, индексируются неотрицательными целыми. Ассоциативные массивы индексируются строками. Заметьте также, что в JavaScript не поддерживаются многомерные массивы (только в виде массивов массивов). И наконец, поскольку JavaScript является нетипизированным языком, элементы массива не обязательно должны иметь одинаковый тип, как в типизированных языках, подобных Java. Мы подробнее поговорим о массивах в главе 9.

3.6.1. Создание массивов

Массивы могут быть созданы с помощью функции-конструктора `Array()`. Созданному массиву может быть присвоено любое количество индексированных элементов:

```
var a = new Array();
a[0] = 1.2;
a[1] = "JavaScript";
a[2] = true;
a[3] = { x:1, y:3 };
```

Массивы могут также быть инициализированы путем передачи элементов массива конструктору `Array()`. Таким образом, предыдущий пример создания и инициализации массива можно записать и так:

```
var a = new Array(1.2, "JavaScript", true, { x:1, y:3 });
```

Если передать конструктору `Array()` только одно число, то оно будет указывать длину массива. Таким образом, код

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

создает новый массив с 10 неопределенными элементами.

3.6.2. Литералы массивов

В ECMAScript v3 определяется (а в JavaScript 1.2 реализуется) синтаксис литералов для создания и инициализации массивов. Литерал массива (или инициализатор массива) – это список разделенных запятыми значений, заключенных в квадратные скобки. Значения в скобках последовательно присваиваются элементам массива с индексами, начиная с нуля.[1] Например, в JavaScript 1.2 код создания и инициализации массива из предыдущего раздела может выглядеть следующим образом:

```
var a = [1.2, "JavaScript", true, { x:1, y:3 }];
```

Как и объектные литералы, литералы массивов могут быть вложенными:

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

Как и в объектных литералах, элементы в литерале массива могут быть произвольными выражениями и не должны ограничиваться константами:

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

Для того чтобы включить в литерал массива неопределенный элемент, достаточно пропустить значение между запятыми. Следующий массив содержит пять элементов, в том числе три неопределенных:

```
var sparseArray = [1,,,,5];
```

3.7. null

Ключевое слово `null` в JavaScript имеет специальный смысл. Обычно считается, что оно имеет объектный тип и говорит об отсутствии объекта. Значение `null` уникально и отличается от любых других. Если переменная равна `null`, следовательно, в ней не содержится допустимый объект, массив, число, строка или логическое значение.[2]

3.8. undefined

Еще одно специальное значение, иногда используемое в JavaScript, – значение `undefined`, возвращаемое при обращении либо к переменной, которая была объявлена, но которой никогда не присваивалось значение, либо к свой-

---

[1] Реализация Netscape JavaScript 1.2 содержит ошибку: когда литерал массива задается с единственным элементом числового типа, это число задает длину массива, а не значение первого элемента. Хотя это поведение соответствует поведению конструктора Array(), в данном контексте оно явно неприемлемо.

[2] Программистам на C и C++ следует обратить внимание, что null в JavaScript – это не то же самое, что 0, как в других языках. В определенных обстоятельствах null преобразуется в 0, однако эти два значения не эквивалентны.

ству объекта, которое не существует. Заметьте, что специальное значение undefined – это не то же самое, что null.

Хотя null и undefined не эквивалентны друг другу, оператор равенства == считает их равными. Рассмотрим следующий код:

```
my.prop == null
```

Это сравнение истинно либо если свойство my.prop не существует, либо если оно существует, но содержит значение null. Поскольку значение null и undefined обозначают отсутствие значения, это равенство часто оказывается тем, что нам нужно. Однако когда действительно требуется отличить значение null от значения undefined, нужен оператор идентичности === или оператор typeof (подробности см. в главе 5).

В отличие от null, undefined не является зарезервированным словом JavaScript. Стандарт ECMAScript v3 указывает, что всегда существует глобальная переменная по имени undefined, начальным значением которой является undefined. Следовательно, в реализации, соответствующей стандарту, можно рассматривать undefined как ключевое слово, если только этой глобальной переменной не присвоено другое значение.

Если нельзя с уверенностью сказать, есть ли в данной реализации переменная undefined, можно просто объявить свою собственную:

```
var undefined;
```

Объявив, но не инициализировав переменную, вы гарантируете, что переменная имеет значение undefined. Оператор void (см. главу 5) предоставляет еще один способ получения значения undefined.

3.9. Объект Date

В предыдущих разделах мы описали все фундаментальные типы данных, поддерживаемые JavaScript. Значения даты и времени не относятся к этим фундаментальным типам, однако в JavaScript имеется класс объектов, представляющих дату и время, который может использоваться для работы с этим типом данных. Объект Date в JavaScript создается с помощью оператора new и конструктора Date() (оператор new будет введен в главе 5, а в главе 8 вы больше узнаете о создании объектов).

```
var now = new Date();  // Создание объекта, в котором хранится текущая дата и
                       // время.
// Создание объекта, в котором хранится дата Рождества.
// Заметьте, что номера месяцев начинаются с нуля, поэтому декабрь имеет номер 11!
var xmas = new Date(2000, 11, 25);
```

Методы объекта Date позволяют получать и устанавливать различные значения даты и времени и преобразовывать дату в строку с использованием либо локального времени, либо времени по Гринвичу (GMT). Например:

```
xmas.setFullYear(xmas.getFullYear() + 1); // Меняем дату на дату следующего
                                          // Рождества.
var weekday = xmas.getDay();  // В 2001 году Рождество попадает на вторник.
document.write("Сегодня: " + now.toLocaleString());  // Текущие дата и время.
```

В объекте Date также определяются функции (не методы, поскольку они не вызываются через объект Date), для преобразования даты, заданной в строковой или числовой форме во внутреннее представление в миллисекундах, полезное для некоторых видов операций с датами.

Полное описание объекта Date и его методов вы найдете в «Справочнике по базовому JavaScript».

3.10. Регулярные выражения

Регулярные выражения предоставляют богатый и мощный синтаксис для описания текстовых шаблонов. Они применяются для поиска соответствия заданному шаблону и реализации операций поиска и замены. В JavaScript для формирования регулярных выражений принят синтаксис языка Perl. Поддержка регулярных выражений впервые была добавлена в версии JavaScript 1.2, а затем стандартизована и расширена в ECMAScript v3.

Регулярные выражения представляются в JavaScript объектом RegExp и могут создаваться с помощью конструктора `RegExp()`. Как и объект Date, объект RegExp не является одним из фундаментальных типов данных JavaScript; это лишь стандартизованный тип объектов, предоставляемый всеми соответствующими реализациями JavaScript.

Однако в отличие от объекта Date, объекты RegExp имеют синтаксис литералов и могут задаваться непосредственно в коде программы на JavaScript 1.2. Текст между парой символов прямой косой черты образует литерал регулярного выражения. За второй косой чертой в паре также могут следовать одна или несколько букв, изменяющих смысл шаблона. Например:

```
/^HTML/
/[1-9][0-9]*/
/\bjavascript\b/i
```

Грамматика регулярных выражений сложна и подробно описана в главе 10. А сейчас вам важно лишь знать, каким литерал регулярного выражения выглядит в коде JavaScript.

3.11. Объекты Error

В ECMAScript v3 определяется несколько классов для представления ошибок. При возникновении ошибки времени выполнения интерпретатор JavaScript «генерирует» объект одного из этих типов. (См. обсуждение вопросов генерации и перехвата ошибок в главе 6 в описании инструкций `throw` и `try`.) Каждый объект ошибки имеет свойство `message`, которое содержит зависящее от реализации сообщение об ошибке. Заранее определены следующие

типы объектов ошибок – Error, EvalError, RangeError, ReferenceError, SyntaxError, TypeError и URIError. Подробнее об этих классах рассказано в справочном разделе, посвященном базовому языку.

3.12. Объекты-обертки для базовых типов данных

Ранее в этой главе мы обсуждали строки, и тогда я обратил ваше внимание на странную особенность этого типа данных: для работы со строками используется объектная нотация. Например, типичная операция со строками может выглядеть следующим образом:

```
var s = "These are the times that try people's souls.";
var last_word = s.substring(s.lastIndexOf(" ")+1, s.length);
```

Если бы вы не знали, то могли бы подумать, что `s` – это объект, и вы вызываете методы и читаете значения свойств этого объекта.

Что же происходит? Являются ли строки объектами или базовыми типами данных? Оператор `typeof` (см. главу 5) убеждает нас, что строки имеют тип данных string, отличный от типа данных object. Почему же тогда для манипуляций со строками используется объектная нотация?

Дело в том, что для каждого из трех базовых типов данных определен соответствующий класс объектов. То есть помимо поддержки типов данных number, string и boolean JavaScript также поддерживает классы Number, String и Boolean. Эти классы представляют собой «обертки» для базовых типов данных. *Обертка* (*wrapper*) содержит такое же значение базового типа, но кроме этого определяет еще свойства и методы, которые могут использоваться для манипуляций с этим значением.

JavaScript может гибко преобразовывать один тип в другой. Когда мы используем строку в объектном контексте, то есть когда пытаемся обратиться к свойству или методу строки, JavaScript создает внутри себя объект-обертку для строкового значения. Этот объект String используется вместо базового строкового значения. Для объекта определены свойства и методы, поэтому удается использовать значение базового типа в объектном контексте. То же самое, конечно, верно и для других базовых типов и соответствующих им объектов-оберток; мы просто не работаем с другими типами в объектном контексте так же часто, как со строками.

Заметьте, что объект String, созданный при использовании строки в объектном контексте, временный – он служит для того, чтобы обеспечить доступ к свойству или методу, после чего необходимость в нем отпадает, и потому он утилизируется системой. Предположим, что `s` – это строка, и мы определяем длину строки с помощью следующего предложения:

```
var len = s.length;
```

Здесь s остается строкой и ее исходное значение не меняется. Создается новый временный объект String, позволяющий обращаться к свойству length, а затем этот объект удаляется, не меняя исходное значение переменной s. Если эта схема кажется элегантной и в то же время неестественно сложной, вы правы. Однако обычно реализации JavaScript выполняют внутреннее преобразование очень эффективно, и вам не стоит об этом беспокоиться.

Для того чтобы явно использовать объект String в своей программе, надо создать постоянный объект, который не будет удален системой автоматически. Объекты String создаются так же, как и другие объекты, – с помощью оператора new. Например:

```
var s = "hello world";              // Значение базового строкового типа
var S = new String("Hello World");  // Объект String
```

Что же можно делать с созданным объектом S типа String? Ничего такого, что нельзя сделать с соответствующим значением базового типа. Если мы воспользуемся оператором typeof, он сообщит нам, что S – это объект, а не строковое значение, но кроме этого, мы не увидим различий между базовым строковым значением и объектом String.[1] Как мы уже видели, строки автоматически преобразуются в объекты String, когда это требуется. Оказывается, что обратное тоже верно. Когда мы используем объект String там, где предполагается значение базового строкового типа, JavaScript автоматически преобразует объект String в строку. Поэтому, если мы используем наш объект String с оператором +, для выполнения операции конкатенации создается временное значение базового строкового типа:

```
msg = S + '!';
```

Имейте в виду, все, что мы говорили в этом разделе о строковых значениях и объектах String, также применимо к числовым и логическим значениям и соответствующим объектам Number и Boolean. Более подробную информацию об этих классах можно получить из соответствующих статей «Справочника по базовому JavaScript». В главе 11 содержатся дополнительные сведения о двойственности «базовый тип/объект» и об автоматическом преобразовании данных в JavaScript.

---

[1] Заметьте, однако, что метод eval() рассматривает строковые значения и объекты String по-разному, и он не будет вести себя так, как вы предполагаете, если вы непреднамеренно передадите ему объект String вместо значения базового строкового типа.

4

Переменные

Переменная – это имя, связанное со значением. Мы говорим, что значение хранится или содержится в переменной. Переменные позволяют хранить данные в программе и работать с ними. Например, следующая строка кода JavaScript присваивает значение `2` переменной с именем `i`:

```
i = 2;
```

А следующая – добавляет `3` к значению переменной `i` и присваивает результат новой переменной `sum`:

```
var sum = i + 3;
```

Это почти все, что надо знать о переменных. Но для полного понимания механизма их работы в JavaScript следует освоить некоторые другие понятия, и пары строк кода здесь недостаточно! В данной главе рассмотрены типизация, объявление, область действия, содержимое и разрешение имен переменных, сбор мусора и двойственность «переменная/свойство».[1]

4.1. Типизация переменных

Самое важное различие между JavaScript и такими языками, как Java и C, состоит в том, что JavaScript *нетипизированный* (*untyped*) язык. В частности, это значит, что переменная JavaScript может содержать значение любого типа, в отличие от переменной Java или C, в которой может содержаться только определенный тип данных, заданный в ее объявлении. Так, в Ja-

---

[1] Это сложный предмет, полное понимание которого требует хорошего знакомства с материалом последующих глав книги. Начинающие могут прочесть первые два раздела и перейти к главам 5, 6 и 7, а затем вернуться к этой главе.

vaScript можно присвоить переменной число, а затем присвоить той же переменной строку:

```
i = 10;
i = "десять";
```

В Java, C, C++ и любом другом строго типизированном языке подобный код недопустим.

Особенность JavaScript, связанная с отсутствием типизации, состоит в том, что язык в случае необходимости легко и автоматически преобразует значения из одного типа в другой. Например, если вы попытаетесь дописать число к строке, JavaScript автоматически преобразует число в соответствующую строку, которая может быть добавлена. Более подробно преобразования типов рассматриваются в главе 11 .

Нетипизированность языка JavaScript обусловливает его простоту по сравнению с типизированными языками, таких как C++ и Java, преимущество которых состоит в том, что они способствуют более строгой практике программирования, облегчая написание, поддержку и повторное использование длинных и сложных программ. Но многие программы на JavaScript представляют собой короткие сценарии, поэтому такая строгость необязательна, и программисты могут воспользоваться преимуществами более простого синтаксиса.

4.2. Объявление переменных

Прежде чем использовать переменную в JavaScript, ее необходимо *объявить*.[1] Переменные объявляются с помощью ключевого слова `var` следующим образом:

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

Можно объявить несколько переменных:

```
var i, sum;
```

Кроме того, объявление переменной можно совмещать с ее инициализацией:

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

Если начальное значение не задано в инструкции `var`, то переменная объявляется, но ее начальное значение остается неопределенным (`undefined`), пока не будет изменено программой.

---

[1] Если этого не сделать, то переменную объявит JavaScript – неявно.

Обратите внимание, что инструкция var также может включаться в циклы for и for/in (см. главу 6), что позволяет объявлять переменную цикла непосредственно в самом цикле. Например:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");
for(var i in o) document.write(i, "<br>");
```

Переменные, объявленные с помощью var, называются *долговременными* (*permanent*): попытка удалить их с помощью оператора delete приведет к ошибке. (Оператор delete рассмотрен в главе 5).

4.2.1. Повторные и опущенные объявления

С помощью инструкции var можно объявить одну и ту же переменную несколько раз. Если повторное объявление содержит инициализатор, то оно действует как обычная инструкция присваивания.

Если попытаться прочитать значение необъявленной переменной, JavaScript сгенерирует сообщение об ошибке. Если присвоить значение переменной, не объявленной с помощью var, JavaScript неявно объявит эту переменную за вас. Заметьте, однако, что переменные, объявленные таким образом, всегда создаются как глобальные, даже если они работают только в теле функции. Чтобы не создавать глобальную переменную (или не использовать существующую) всегда помещайте инструкцию var в тело функции, если хотите создать локальную переменную для отдельной функции. Лучше всего объявлять с ключевым словом var все переменные – и глобальные и локальные. (Различие между локальными и глобальными переменными подробнее рассматривается в следующем разделе.)

4.3. Область действия переменной

Область действия (*scope*) переменной – это раздел программы, для которого эта переменная определена. *Глобальная* переменная имеет глобальную область действия – она определена для всей программы на JavaScript. Переменные, объявленные внутри функции, определены только в ее теле. Это *локальные* переменные, и они имеют локальную область действия. Параметры функции также считаются локальными переменными и определены только в теле этой функции.

Внутри тела функции локальная переменная имеет преимущество перед глобальной переменной с тем же именем. Если объявить локальную переменную или параметр функции с тем же именем, что у глобальной переменной, то фактически глобальная переменная будет скрыта. Так, следующий код печатает слово «локальная»:

```
var scope = "глобальная";       // Объявление глобальной переменной
function checkscope() {
```

```
    var scope = "локальная";   // Объявление локальной переменной с тем же именем
    document.write(scope);     // Используется локальная переменная,
                               // а не глобальная
}
checkscope();                  // Печатается "локальная"
```

Объявляя переменные с глобальной областью действия, инструкцию var можно опустить, но при объявлении локальных переменных она необходима. Посмотрите, что получается, если этого не сделать:

```
scope = "глобальная";          // Объявление глобальной переменной, даже без var
function checkscope() {
    scope = "локальная";       // Ой! Мы только что изменили глобальную переменную
    document.write(scope);     // Используется глобальная переменная
    myscope = "локальная";     // Здесь мы неявно объявляем новую глобальную
                               // переменную
    document.write(myscope);   // Используется новая глобальная переменная
}
checkscope();                  // Печает "локальнаялокальная"
document.write(scope);         // Печатает "локальная"
document.write(myscope);       // Печатает "локальная"
```

Функции, как правило, не знают, какие переменные объявлены в глобальной области действия или для чего они нужны. Поэтому функция, использующая глобальную переменную вместо локальной, рискует изменить значение, необходимое какой-либо другой части программы. К счастью, избежать этой неприятности легко: объявляйте все переменные с помощью var.

В JavaScript 1.2 (и в ECMAScript v3) определения функций могут быть вложенными. Каждая функция имеет собственную локальную область действия, и поэтому может быть несколько вложенных уровней локальных областей действия. Например:

```
var scope = "глобальная область действия";          // Глобальная переменная
function checkscope() {
    var scope = "локальная область действия";       // Локальная переменная
    function nested() {
        var scope = "вложенная область действия";  // Вложенная область действия
                                                   // локальных переменных
        document.write(scope);       // Печатает "вложенная область действия"
    }
    nested();
}
checkscope();
```

4.3.1. Отсутствие блочной области действия

Обратите внимание, что в отличие от C, C++ и Java, в JavaScript нет области действия на уровне блоков. Все переменные, объявленные внутри функции, независимо от того, где именно это сделано, определены *во всей* функции.

В следующем коде переменные i, j и k имеют одинаковые области действия: все три определены во всем теле функции. Это было бы не так, если бы код был написан на C, C++ или Java:

```
function test(o) {
    var i = 0;                          // i определена во всей функции
    if (typeof o == "object") {
        var j = 0;                      // j определена везде, а не только в блоке
        for(var k = 0; k < 10; k++) {   // k определена везде, а не только в цикле
            document.write(k);
        }
        document.write(k);              // k все еще определена: печатается 10
    }
    document.write(j);                  // j определена, но может быть
                                        // не инициализирована
}
```

Правило, согласно которому все переменные, объявленные в функции, определены во всей функции, может иметь удивительные следствия. Например:

```
var scope = "глобальная";
function f() {
    alert(scope);              // Показывает "undefined", а не "глобальная".
    var scope = "локальная";   // Переменная инициализируется здесь,
                               // но определена она везде.
    alert(scope);              // Показывает "локальная"
}
f();
```

Кто-то может подумать, что в результате первого вызова alert() будет напечатано слово «глобальная», так как инструкция var, объявляющая локальную переменную, еще не была выполнена. Однако согласно правилу определения областей действия все происходит не так. Локальная переменная определена во всем теле функции, значит, глобальная переменная с тем же именем скрыта от всего тела функции. Хотя локальная переменная определена везде, до выполнения инструкции var она не инициализирована. Поэтому функция f в предыдущем примере эквивалентна следующему:

```
function f() {
    var scope;             // Локальная переменная определяется в начале функции
    alert(scope);          // Здесь она существует, но имеет значение «undefined»
    scope = "локальная";   // Здесь мы инициализируем переменную и присваиваем ей
                           // значение
    alert(scope);          // Здесь она уже имеет значение
}
```

Этот пример показывает, почему хорошая практика программирования подразумевает помещение всех объявлений переменных в начале функции.

4.3.2. Неопределенные и неинициализированные переменные

Примеры предыдущего раздела демонстрируют тонкий момент программирования на JavaScript: имеется два вида неопределенных переменных. Первый – переменная, которая никогда не была объявлена. Попытка прочитать значение такой необъявленной переменной приведет к ошибке времени выполнения. *Необъявленные переменные* не определены, потому что они просто не существуют. Как уже было сказано, присваивание значения необъявленной переменной не приводит к ошибке, – просто она неявно объявляется как глобальная переменная.

Второй вид – переменная, которая была объявлена, но которой никогда не присваивалось значение. Если прочитать значение одной из таких переменных, то будет получено ее значение по умолчанию – `undefined`. Такие переменные лучше называть *неинициализированными* (*unassigned*), чтобы отличить их от тех переменных, которые даже не были объявлены и не существуют.

Следующий фрагмент кода иллюстрирует некоторые различия между неопределенными и неинициализированными переменными:

```
var x;      // Объявляем неинициализированную переменную. Значением ее
            // является undefined.
alert(u);   // Использование необъявленной переменной приведет к ошибке.
u = 3;      // Присваивание значения необъявленной переменной создает
            // эту переменную.
```

4.4. Базовые и ссылочные типы

Следующая тема, которую мы рассмотрим – содержимое переменных. Мы часто говорим, что переменная содержит значение. Что же она содержит в действительности? Чтобы ответить на этот, казалось бы, простой вопрос, мы должны снова взглянуть на типы данных, поддерживаемые JavaScript. Эти типы можно разделить на две группы: базовые типы и ссылочные типы. Числа, логические значения, `null` и `undefined` – это базовые типы. Объекты, массивы и функции – это ссылочные типы.

Базовый тип имеет фиксированный размер. Например, число занимает восемь байтов, а логическое значение представлено всего одним битом. Числовой тип – самый большой из базовых типов. Если для каждой переменной JavaScript зарезервировано в памяти восемь байт, переменная может непосредственно содержать значение любого базового типа.[1]

Но ссылочные типы – это другое дело. Например, объекты могут быть любой длины – они не имеют фиксированного размера. То же самое относится и к массивам: массив может иметь любое число элементов. Аналогично, функция

[1] Это упрощение, которое не следует рассматривать как описание фактической реализации JavaScript.

может содержать любой объем кода JavaScript. Поскольку эти типы не имеют фиксированного размера, их значения не могут непосредственно храниться в восьми байтах памяти, связанных с каждой переменной. Поэтому в переменной хранится *ссылка* на это значение. Обычно эта ссылка представляет собой какой-либо указатель или адрес в памяти. Ссылка – это не само значение, но она сообщает переменной, где это значение можно найти.

Различие между базовыми и ссылочными типами существенно, так как они ведут себя по-разному. Рассмотрим следующий код, оперирующий числами (базовый тип):

```
var a = 3.14;  // Объявление и инициализация переменной
var b = a;     // Копирование значения переменной в новую переменную
a = 4;         // Модификация значения исходной переменной
alert(b)       // Показывает 3.14; копия не изменилась
```

В этом коде нет ничего удивительного. Теперь посмотрим, что произойдет, если мы слегка изменим код, заменив числа массивами (ссылочный тип):

```
var a = [1,2,3];  // Инициализируем переменную ссылкой на массив
var b = a;        // Копируем эту ссылку в новую переменную
a[0] = 99;        // Изменяем массив, используя первоначальную ссылку
alert(b);         // Показываем измененный массив [99,2,3], используя новую ссылку
```

Те, кого результат не удивил, уже хорошо знакомы с различием между базовыми и ссылочными типами. Тем же, кого он удивил, надо посмотреть внимательнее на вторую строку. Обратите внимание, что в этом предложении выполняется присваивание ссылки на значение типа «массив», а не присваивание самого массива. После второй строки кода мы все еще имеем один объект массива; нам только удалось получить две ссылки на него.

Если разница между базовым и ссылочным типами вам внове, просто постарайтесь держать в уме содержимое переменной. Переменные содержат фактические значения базовых типов, но лишь ссылки на значения ссылочных типов. Разное поведение базовых и ссылочных типов более подробно изучается в разделе 11.2 «По значению и по ссылке».

Вы могли заметить, что я не указал, относятся ли строки в JavaScript к базовым или к ссылочным типам. Строки – это необычный случай. Они имеют переменную длину и потому, очевидно, не могут храниться непосредственно в переменных фиксированного размера. Исходя из соображений эффективности, можно ожидать, что JavaScript будет копировать ссылки на строки, а не их фактическое содержимое. С другой стороны, во многих отношениях строки ведут себя, как базовые типы. Вопрос о том, к какому типу принадлежат строки – к базовому или к ссылочному, спорный, так как строки на самом деле *неизменны*: нет возможности изменить содержимое строкового значения. Это значит, что нельзя составить пример, похожий на предыдущий, в котором массивы копировались по ссылке. В конце концов, не имеет значения, как рассматривать строки – как неизменный ссылочный тип, ведущий себя, как базовый, или как базовый тип, реализованный с использованием механизма ссылочного типа.

4.5. Сборка мусора

Ссылочные типы не имеют фиксированного размера; в самом деле некоторые из них могут быть очень большими. Мы уже говорили о том, что переменные не содержат непосредственных значений ссылочного типа. Значения хранятся в каком-либо другом месте, а в переменных находится только ссылка на это местоположение. А сейчас кратко остановимся на реальном хранении значений.

Поскольку строки, объекты и массивы не имеют фиксированного размера, место для их хранения должно выделяться динамически, когда становится известен размер. Когда программа на JavaScript создает строку, массив или объект, интерпретатор должен выделить память для хранения этой сущности. Память, выделяемая подобным образом, должна быть впоследствии освобождена, иначе интерпретатор JavaScript исчерпает всю доступную память, и это приведет к отказу системы.

В таких языках, как C и C++, память приходится освобождать вручную. Именно программист отвечает за отслеживание всех создаваемых объектов и, когда они больше не требуются, за их ликвидацию (освобождение памяти). Это бывает довольно обременительно и часто приводит к ошибкам.

В JavaScript, где не надо вручную освобождать память, реализована технология, называемая *сборкой мусора* (*garbage collection*). Интерпретатор JavaScript может обнаружить, что объект никогда более не будет использоваться программой. Определив, что объект недоступен (то есть больше нет способа получения ссылки на него), интерпретатор знает, что объект более не нужен, и занятая им память может быть освобождена. Рассмотрим следующие строки кода:

```
var s = "hello";          // Выделяем память для строки
var u = s.toUpperCase();  // Создаем новую строку
s = u;                    // Переписываем ссылку на первоначальную строку
```

После работы этого кода исходная строка «hello» больше недоступна – ни в одной из переменных программы нет ссылки на нее. Система определяет этот факт и освобождает память.

Сборка мусора выполняется автоматически и невидима для программиста. Программист может создавать столько объектов, сколько ему надо, и система выполнит очистку сама! О сборке мусора он должен знать ровно столько, сколько ему требуется, чтобы доверять ее работе, – он не должен думать, куда делись все старые объекты. Однако для тех, кого это не устраивает, в разделе 11.3 «Сборка мусора» приводятся более детальные сведения о сборке мусора в JavaScript.

4.6. Переменные как свойства

Вы уже могли заметить, что в JavaScript между переменными и свойствами объектов много общего. Им одинаково присваиваются значения, они одинаково используются в выражениях JavaScript и т. д. Есть ли на самом деле какая-нибудь принципиальная разница между переменной `i` и свойством `i` объекта `o`? Ответ – никакой. Переменные в JavaScript принципиально не отличаются от свойств объекта.

4.6.1. Глобальный объект

Одно из первых действий, выполняемых интерпретатором JavaScript при запуске, перед исполнением любого кода, – это создание *глобального объекта*. Свойства этого объекта представляют собой глобальные переменные программы на JavaScript. Объявляя глобальную переменную JavaScript, фактически вы определяете свойство глобального объекта.

Интерпретатор JavaScript инициализирует ряд свойств глобального объекта, ссылающихся на предопределенные значения и функции. Так, свойства `Infinity`, `parseInt` и `Math` ссылаются на число «бесконечность», предопределенную функцию `parseInt()` и предопределенный объект Math. Об этих глобальных значениях рассказано в справочном разделе этой книги, посвященном базовому языку.

В коде верхнего уровня (то есть коде JavaScript, который не является частью функции) ссылка на глобальный объект может осуществляться посредством ключевого слова JavaScript `this`. Внутри функций ключевое слово `this` имеет другое применение, которое описано в главе 7.

В клиентском JavaScript в качестве глобального объекта для всего кода JavaScript, содержащегося в соответствующем ему окне броузера, служит объект Window. Этот глобальный объект Window имеет свойство `window`, ссылающееся на сам объект, которое можно использовать вместо `this` для ссылки на глобальный объект. Объект Window определяет базовые глобальные свойства, такие как `parseInt` и `Math`, а также глобальные клиентские свойства, такие как `navigator` и `screen`.

4.6.2. Локальные переменные: объект вызова

Если глобальные переменные – это свойства специального глобального объекта, то что же тогда такое локальные переменные? Они тоже представляют собой свойства объекта. Этот объект называется *объектом вызова* (*call object*). Когда выполняется тело функции, аргументы и локальные переменные функции хранятся как свойства этого объекта. Использование абсолютно отдельного объекта для локальных переменных позволяет JavaScript не допускать переписывания локальными переменными значений глобальных переменных с теми же именами.

4.6.3. Контексты исполнения JavaScript

Начиная исполнять функцию, интерпретатор JavaScript создает для нее новый *контекст исполнения* (*execution context*). Контекст исполнения – это контекст, в котором выполняется любой фрагмент кода JavaScript. Важная часть контекста – объект, в котором определены переменные. Поэтому код JavaScript, не являющийся частью какой-либо функции, работает в контексте исполнения, в котором для определений переменных используется глобальный объект. А любая функция JavaScript работает в своем собственном уникальном контексте исполнения со своим собственным объектом вызова, в котором определены локальные переменные.

Интересно отметить, что реализации JavaScript могут допускать несколько глобальных контекстов исполнения, каждый – с *отдельным* глобальным объектом. (Хотя в этом случае каждый глобальный объект не является действительно глобальным.)[1] Очевидный пример – это клиентский JavaScript, в котором каждое отдельное окно броузера или каждый фрейм в окне определяет отдельный глобальный контекст исполнения. Код клиентского JavaScript в каждом фрейме или окне работает в своем собственном контексте исполнения и имеет свой собственный глобальный объект. Однако эти отдельные клиентские глобальные объекты имеют свойства, связывающие их друг с другом. Другими словами, код JavaScript в одном фрейме может ссылаться на другой фрейм с помощью выражения `parent.frames[1]`, а на глобальную переменную `x` в первом фрейме можно сослаться из второго фрейма с помощью выражения `parent.frames[0].x`.

Вы не должны уже сейчас полностью понимать, как связываются вместе контексты исполнения отдельных окон и фреймов в клиентском JavaScript. Эту тему мы подробно рассмотрим при обсуждении интеграции JavaScript с веб-броузерами в главе 12. Сейчас достаточно знать, что гибкость JavaScript позволяет одному интерпретатору JavaScript исполнять сценарии в различных глобальных контекстах исполнения, и что эти контексты не обязаны быть совершенно раздельными – они могут ссылаться друг на друга.

Последнее утверждение надо рассмотреть подробнее. Если код JavaScript в одном контексте исполнения может читать и писать значения свойств и выполнять функции, определенные в другом контексте исполнения, то становятся актуальными вопросы безопасности. Возьмем в качестве примера клиентский JavaScript. Предположим, что окно броузера A запускает сценарий или содержит информацию из вашей локальной сети, а окно B запускает сценарий с некоторого произвольного сайта в Интернете. Скорее всего, мы не захотим предоставлять коду в окне B доступ к свойствам в окне A. Ведь тогда этот код получит возможность прочитать важную корпоративную информацию и, например, украсть ее. Следовательно, безопасный запуск кода JavaScript должен обеспечить специальный механизм, предотвращающий

[1] Это отход от темы, если он вам неинтересен, спокойно переходите к следующему разделу.

доступ из одного контекста исполнения в другой, когда этот доступ не должен быть разрешен. Мы вернемся к этой теме в главе 21.

4.7. Еще об области действия переменных

Когда мы в первый раз обсуждали понятие области действия переменной, я определил его только на основе лексической структуры кода JavaScript: глобальные переменные имеют глобальную область действия, а переменные, объявленные в функции, – локальную. Если одно определение функции вложено в другое, то переменные, объявленные в этой вложенной функции, имеют вложенную локальную область действия. Теперь, когда мы знаем, что глобальные переменные представляют собой свойства глобального объекта, а локальные – свойства специального объекта вызова, мы можем вернуться к понятию области действия переменной и переосмыслить его. Это новое описание области действия предоставляет нам полезную возможность по-новому взглянуть на существование переменных во многих контекстах и глубже понять, как работает JavaScript.

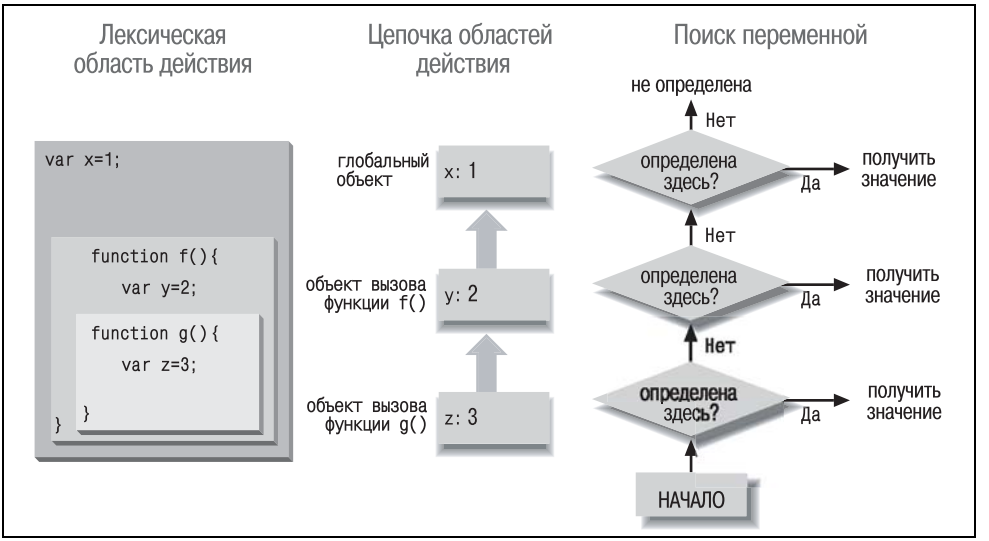

Рис. 4.1. *Цепочка областей действия и разрешения имени переменной*

С каждым контекстом исполнения JavaScript связана *цепочка областей действия* (*scope chains*). Эта цепочка областей действия представляет собой список или цепочку объектов. Когда коду JavaScript требуется найти значение переменной *x* (этот процесс называется *разрешением имени переменной*), он начинает поиск в первом объекте цепочки. Если в этом объекте отыскивается свойство с именем *x*, то используется значение этого свойства. Если в первом объекте не удается найти свойство с именем *x*, то JavaScript продолжает поиск в следующем объекте цепочки. Если во втором объекте

тоже не найдено свойство с именем *x*, поиск продолжается в следующем объекте, и т. д.

В коде JavaScript верхнего уровня (в коде, не содержащемся ни в одном из определений функций), цепочка областей действия состоит только из глобального объекта. Все переменные разыскиваются в этом объекте. Если переменная не существует, то ее значение равно `undefined`. В функции (не вложенной) цепочка областей действия состоит из двух объектов. Когда функция ссылается на переменную, в первую очередь проверяется объект вызова (локальная область действия), а во вторую очередь – глобальный объект (глобальная область действия). Вложенная функция будет иметь три или более объектов в цепочке областей действия. Процесс поиска имени переменной в цепочке областей действия функции иллюстрирует рис. 4.1.

5

Выражения и операторы

В этой главе объясняется, как работают выражения и операторы в JavaScript. Те, кто знаком с C, C++ или Java, заметят, что выражения и операторы в JavaScript очень похожи, и смогут ограничиться беглым просмотром этой главы. А те, кто не программирует на C, C++ или Java, в этой главе узнают все, что требуется знать о выражениях и операторах в JavaScript.

5.1. Выражения

Выражение – это фраза JavaScript, которая может быть *вычислена* интерпретатором для получения значения. Простейшие выражения – это литералы или имена переменных, например:

```
1.7                      // Числовой литерал
"JavaScript is fun!"     // Строковый литерал
true                     // Логический литерал
null                     // Литерал значения null
/java/                   // Литерал регулярного выражения
{ x:2, y:2 }             // Объектный литерал
[2,3,5,7,11,13,17,19]    // Литерал массива
function(x){return x*x;} // Функциональный литерал
i                        // Переменная i
sum                      // Переменная sum
```

Значение выражения-литерала – это просто значение самого литерала. Значение выражения-переменной – это значение, содержащееся в переменной, или то, на которое переменная ссылается.

Эти выражения не очень интересны. Путем объединения простых выражений могут создаваться более сложные (и интересные) выражения. Напри-

мер, мы видели, что `1.7` – это выражение, и `i` – это выражение. Следующий пример тоже представляет собой выражение:

```
i + 1.7
```

Значение этого выражения определяется путем сложения значений двух более простых выражений. Знак + в этом примере – это *оператор*, объединяющий два выражения в одно более сложное. Другой оператор это – (минус), объединяющий выражения путем вычитания. Например:

```
(i + 1.7) - sum
```

В этом выражении оператор «минус» применяется для вычитания значения переменой `sum` из значения предыдущего выражения, `i + 1.7`. Как вы увидите в следующем разделе, JavaScript поддерживает несколько других операторов кроме + и –.

5.2. Обзор операторов

Если вы программировали на C, C++ или Java, то большинство операторов JavaScript должны быть уже вам знакомы. Они сведены в табл. 5.1, к которой можно обращаться как к справочнику. Заметьте, что большинство операторов обозначаются символами пунктуации, такими как + и =, а некоторые – ключевыми словами, например `delete` и `instanceof`. И ключевые слова и знаки пунктуации обозначают обычные операторы, просто первый способ дает более удобочитаемый и менее лаконичный синтаксис.

В этой таблице столбец, обозначенный буквой «P», содержит приоритет оператора, а столбец, обозначенный буквой «A», – ассоциативность оператора (либо L – слева направо, либо R – справа налево). Те, кто пока не понимает, что это такое, получат разъяснения в следующих подразделах, после которых приводятся описания самих операторов.

Таблица 5.1. Операторы JavaScript

P	A	Оператор	Типы операндов	Выполняемая операция
15	L	`.`	объект, идентификатор	Обращение к свойству
	L	`[]`	массив, целое	Индексация массива
	L	`()`	функция, аргументы	Вызов функции
	R	`new`	вызов конструктора	Создание нового объекта
14	R	`++`	левое значение	Префиксный или постфиксный инкремент (унарный)
	R	`--`	левое значение	Префиксный или постфиксный декремент (унарный)
	R	`-`	число	Унарный минус (смена знака)

Таблица 5.1 (продолжение)

P	A	Оператор	Типы операндов	Выполняемая операция
	R	+	число	Унарный плюс (нет операции)
	R	~	целое	Битовое дополнение (унарный)
	R	!	логическое	Логическое дополнение (унарный)
	R	delete	левое значение	Аннулирование определения свойства (унарный)
	R	typeof	любой	Возвращает тип данных (унарный)
	R	void	любой	Возвращает неопределенное значение (унарный)
13	L	*, /, %	числа	Умножение, деление, остаток
12	L	+, -	числа	Сложение, вычитание
	L	+	строки	Конкатенация строк
11	L	<<	целые	Сдвиг влево
	L	>>	целые	Сдвиг вправо с расширением знакового разряда
	L	>>>	целые	Сдвиг вправо с дополнением нулями
10	L	<, <=	числа или строки	Меньше чем, меньше или равно
	L	>, >=	числа или строки	Больше чем, больше или равно
	L	instanceof	объект, конструктор	Проверка типа объекта
	L	in	строка, объект	Проверка наличия свойства
9	L	==	любой	Проверка на равенство
	L	!=	любой	Проверка на неравенство
	L	===	любой	Проверка на идентичность
	L	!==	любой	Проверка не неидентичность
8	L	&	целые	Поразрядное И
7	L	^	целые	Поразрядное исключающее ИЛИ
6	L	|	целые	Поразрядное ИЛИ
5	L	&&	логические	Логическое И
4	L	||	логические	Логическое ИЛИ
3	R	?:	логическое, любое, любое	Условный оператор (3 операнда)
2	R	=	левое значение, любое	Присваивание
	R	*=, /=, %=, +=, -=, <<=, >>=, >>>=, &=, ^=, |=	левое значение, любое	Присваивание с операцией
1	L	,	любой	Множественное вычисление

5.2.1. Количество операндов

Операторы могут быть разбиты на категории по количеству требуемых им операндов. Большинство операторов JavaScript, как оператор +, о котором мы уже говорили, представляют собой *бинарные операторы*, которые объединяют два выражения в одно, более сложное. Таким образом, эти операторы работают с двумя операндами. JavaScript также поддерживает несколько *унарных операторов*, которые преобразуют одно выражение в другое, более сложное. Оператор «минус» в выражении `-3` представляет собой унарный оператор, выполняющий смену знака для операнда `3`. И наконец, JavaScript поддерживает один *тернарный оператор*, условный оператор `?:`, который объединяет в одно значения трех выражений.

5.2.2. Тип операндов

Создавая выражения JavaScript, необходимо обращать внимание на типы данных, передаваемых операторам, и на типы данных, которые они возвращают. Различные операторы требуют, чтобы операнды возвращали значения определенного типа. Например, нельзя выполнить умножение строк, поэтому выражение `"a" * "b"` не является допустимым в JavaScript. Заметьте, однако, что JavaScript пытается, если возможно, преобразовать выражение в требуемый тип, поэтому выражение `"3" * "5"` вполне допустимо. Его значением будет число `15`, а не строка `"15"`. Мы подробно рассмотрим преобразование типов в JavaScript в разделе 11.1 «Преобразование типов».

Кроме того, некоторые операторы ведут себя по-разному в зависимости от типа операндов. Самый яркий пример – оператор +, который складывает числовые операнды и выполняет конкатенацию строк. Кроме того, если ему переданы одна строка и одно число, он преобразует число в строку и выполняет конкатенацию двух полученных строк. Например, результатом выражения `"1" + 0` будет строка `"10"`.

Обратите внимание, что операторы присваивания, как и некоторые другие, требуют в качестве выражений в левой части «левого значения» (lvalue). *Левое значение* – это исторический термин, обозначающий «выражение, которое может присутствовать в левой части инструкции присваивания». В JavaScript левыми значениями являются переменные, свойства объектов и элементы массивов. Спецификация ECMAScript разрешает встроенным функциям возвращать левые значения, но не определяет никаких встроенных функций, ведущих себя подобным образом.

И наконец, операторы не всегда возвращают значения того же типа, к которому принадлежат операнды. Операторы сравнения (меньше, равно, больше и т. д.) принимают в качестве аргументов различные типы, но всегда возвращают результат логического типа, показывающий, истинно ли сравнение. Так, выражение `a<3` возвращает `true`, если значение переменной `a` действительно меньше, чем `3`. Как мы увидим, логические значения, возвращаемые операторами сравнения, используются в инструкциях `if`, циклах `while` и `for`, управляющих в JavaScript исполнением программы в зависимости от результатов вычисления выражений с операторами сравнения.

5.2.3. Приоритет операторов

В табл. 5.1, в столбце, помеченном буквой «P», указан *приоритет* каждого оператора. Приоритет оператора управляет порядком, в котором выполняются операции. Операторы с большими числами в столбце «P» выполняются раньше, чем те, для которых указаны меньшие числа.

Рассмотрим следующее выражение:

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

Оператор умножения * имеет больший приоритет по сравнению с оператором сложения +, поэтому умножение выполняется до сложения. Кроме того, оператор присваивания = имеет наименьший приоритет, поэтому присваивание выполняется после завершения всех операций в правой части.

Приоритет операторов может быть переопределен с помощью скобок. Для того чтобы сложение в предыдущем примере выполнялось раньше, надо написать:

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

На практике, если вы не уверены в приоритетах операторов, проще всего явно задать порядок вычислений с помощью скобок. Важно знать лишь следующие правила: умножение и деление выполняются до сложения и вычитания, а присваивание имеет очень низкий приоритет и почти всегда выполняется последним.

5.2.4. Ассоциативность операторов

В табл. 5.1, в столбце, помеченном буквой «A», указана ассоциативность оператора. Значение L задает ассоциативность слева направо, а значение R – ассоциативность справа налево. Ассоциативность оператора определяет порядок выполнения операций с одинаковым приоритетом. Ассоциативность слева направо обозначает, что операции выполняются слева направо. Например, оператор сложения имеет ассоциативность слева направо, поэтому

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

представляет собой то же самое, что и

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

С другой стороны, следующие (практически бессмысленные) выражения

```
x = ~-~y;
w = x = y = z;
q = a?b:c?d:e?f:g;
```

эквивалентны

```
x = ~(-(~y));
```

```
w = (x = (y = z));
q = a?b:(c?d:(e?f:g));
```

так как унарные операторы, операторы присваивания и условные тернарные операторы имеют ассоциативность справа налево.

5.3. Арифметические операторы

Рассказав о приоритетах, ассоциативности и других второстепенных вопросах, мы можем начать обсуждение самих операторов. В этом разделе приведены описания арифметических операторов:

Сложение (+)

Оператор «плюс» складывает числовые операнды или выполняет конкатенацию строк. Если одним из операндов является строка, другой операнд преобразуется в строку и выполняется конкатенация. Операнды-объекты преобразуются в числа или строки, которые могут быть сложены или конкатенированы. Преобразование выполняется с помощью методов `valueOf()` и/или `toString()`.

Вычитание (–)

Когда «минус» используется в качестве бинарного оператора, он выполняет вычитание второго операнда из первого. Если указаны нечисловые операнды, то он пытается преобразовать их в числа.

Умножение (*)

Оператор `*` умножает два своих операнда. Нечисловые операнды он пытается преобразовать в числа.

Деление (/)

Оператор / делит первый операнд на второй. Пытается преобразовать нечисловые операнды в числа. Те, кто привык к языкам программирования, различающим целые и вещественные числа, могут ожидать получения целочисленного результата при делении одного целого на другое. Однако в JavaScript все числа вещественные, поэтому все деления дают результат с плавающей точкой. Операция `5/2` дает `2.5`, а не `2`. Деление на ноль дает в качестве результата плюс или минут бесконечность, а `0/0` дает `NaN`.

Деление по модулю (%)

Оператор `%` вычисляет остаток, получаемый при целочисленном делении первого операнда на второй. Если заданы нечисловые операнды, то оператор пытается преобразовать их в числа. Знак результата совпадает со знаком первого операнда. Например, `5 % 2` дает `1`.

Оператор деления по модулю обычно применятся с целыми операндами, но работает и для вещественных значений. Например, `-4.3 % 2.1` дает результат `-0.1`.

Унарный минус (–)

Когда минус используется в качестве унарного оператора перед одиночным операндом, он выполняет унарную операцию смены знака. Другими словами, он преобразует положительное значение в отрицательное и наоборот. Если операнд не является числом, этот оператор пытается преобразовать его в число.

Унарный плюс (+)

Для симметрии с оператором «унарный минус» в JavaScript также имеется оператор унарный плюс. При помощи этого оператора можно явно задавать знак числовых литералов, если вы считаете, что это сделает текст программы более понятным:

```
var profit = +1000000;
```

В таком коде оператор «плюс» ничего не делает; результатом его работы является значение его аргумента. Однако нечисловые аргументы он преобразует в числа. Если аргумент не может быть преобразован, возвращается `NaN`.

Инкремент (++)

Этот оператор инкрементирует (то есть увеличивает на единицу) свой единственный операнд, который должен быть переменной, элементом массива или свойством объекта. Если значение этой переменной, элемента массива или свойства не является числом, оператор сначала пытается преобразовать его в число. Точное поведение этого оператора зависит от его положения по отношению к операнду. Если поставить его перед операндом (префиксный оператор инкремента), то к операнду прибавляется 1, а результатом является увеличенное значение операнда. Если же он размещен после операнда (постфиксный оператор инкремента), то к операнду прибавляется 1, однако результатом является *первоначальное* значение операнда. Если увеличиваемое значение не является числом, оно в процессе вычисления преобразуется в число.

Например, следующий код устанавливает переменные `i` и `j` равными `2`:

```
i = 1;
j = ++i;
```

А этот устанавливает `i` в `2`, а `j` в `1`:

```
i = 1;
j = i++;
```

Данный оператор в обеих своих формах чаще всего применяется для увеличения счетчика, управляющего циклом. Заметьте, что нельзя вставлять перевод строки между префиксным или постфиксным оператором инкремента и предшествующим ему операндом, поскольку точки с запятой в JavaScript вставляются автоматически. Если это сделать, Java-

Script будет рассматривать операнд как полноценную инструкцию и вставит после него точку с запятой.

Декремент (--)

Этот оператор декрементирует (то есть уменьшает на 1) свой единственный числовой операнд, который может представлять собой переменную, элемент массива или свойство объекта. Если значение этой переменной, элемента или свойства не является числом, оператор сначала пытается преобразовать его в число. Как и для оператора `++`, точное поведение оператора `--` зависит от его положения относительно операнда. Будучи поставленным перед операндом, он уменьшает операнд и возвращает уменьшенное значение. После операнда он уменьшает операнд, но возвращает *первоначальное* значение.

5.4. Операторы равенства

В этом разделе описаны операторы равенства и неравенства в JavaScript. Это операторы, сравнивающие два значения и возвращающие логическое значение (`true` или `false`) в зависимости от результата сравнения. Как мы увидим в главе 6, чаще всего они применяются в инструкциях `if` и циклах `for` для управления ходом исполнения программы.

5.4.1. Равенство (==) и идентичность (===)

Операторы `==` и `===` проверяют две величины на совпадение, руководствуясь двумя определениями совпадения. Оба оператора принимают операнды любого типа и оба возвращают `true`, если их операнды совпадают, и `false`, если они различны. Оператор `===`, известный как оператор идентичности, проверяет два операнда на «идентичность», руководствуясь строгим определением совпадения. Оператор `==` известен как оператор равенства, он проверяет, равны ли два его операнда в соответствии с более слабым определением совпадения, допускающим преобразования типов.

Оператор идентичности стандартизован в ECMAScript v3 и реализован в JavaScript 1.3 и более поздних версиях. С введением оператора идентичности JavaScript стал поддерживать операторы `=`, `==` и `===`. Убедитесь, что вы понимаете разницу между операторами присваивания, равенства и идентичности, будьте внимательны и применяйте правильные операторы при написании кода! Хотя заманчиво называть все три оператора «равно», но лучше, во избежание путаницы, читать `=` как «получается» или «присваивается», `==` читать как «равно» и словом «идентично» обозначать оператор `===`.

В JavaScript числовые, строковые и логические значения сравниваются *по значению*. В этом случае рассматриваются две различные величины, а операторы `==` и `===` проверяют, идентичны ли эти два значения. Это значит, что две переменные равны или идентичны, только если они содержат одинаковое значение. Например, две строки равны, только если обе содержат в точности одинаковые символы.

С другой стороны, объекты, массивы и функции сравниваются *по ссылке*. Это значит, что две переменные равны, только если они *ссылаются* на один и тот же объект. Два отдельных массива никогда не могут быть равными или идентичными, даже если они содержат равные или идентичные элементы. Две переменные, содержащие ссылки на объекты, массивы или функции, равны, только если ссылаются на один и тот же объект, массив или функцию. Для того чтобы проверить, содержат ли два различных объекта одинаковые свойства или содержат ли два различных массива одинаковые элементы, надо проверить равенство или идентичность для каждого свойства или элемента. (И если какое либо свойство или элемент само является объектом или массивом, решить, на какую глубину вложенности вы хотите выполнять сравнение.)

Для определения идентичности двух значений оператор `===` руководствуется следующими правилами:

- Если два значения имеют различные типы, они не идентичны.
- Два значения идентичны, только если оба они представляют собой числа, имеют одинаковые значения и не являются `NaN` (в этом, последнем случае они не идентичны). Значение `NaN` никогда не бывает идентичным никакому значению, даже самому себе! Чтобы проверить, является ли значение `NaN`, используйте глобальную функцию `isNaN()`.
- Если оба значения представляют собой строки и содержат одни и те же символы в тех же позициях, они идентичны. Если строки отличаются по длине или содержимому, они не идентичны. Обратите внимание, что в некоторых случаях стандарт Unicode допускает несколько способов кодирования одной и той же строки. Однако для повышения эффективности сравнение строк в JavaScript выполняется строго посимвольно, при этом предполагается, что все строки перед сравнением преобразованы в «нормализованную форму». Другой способ сравнения строк обсуждается в части III «Справочник по базовому JavaScript» на странице с описанием метода String.localeCompare().
- Если оба значения представляют собой логические значения `true` или `false`, то они идентичны.
- Если оба значения ссылаются на один и тот же объект, массив или функцию, то они идентичны. Если они ссылаются на различные объекты (массивы или функции), они не идентичны, даже если оба имеют идентичные свойства или идентичные элементы.
- Если оба значения равны `null` или `undefined`, то они идентичны.

Следующие правила применяются для определения равенства при помощи оператора `==`:

- Если два значения имеют одинаковый тип, проверяем их на идентичность. Если значения идентичны, они равны; если они не идентичны, они не равны.

- Если два значения не относятся к одному и тому же типу, они все же могут быть равными. Правила и преобразования типов при этом такие:
  - Если одно значение равно `null`, а другое – `undefined`, то они равны.
  - Если одно значение представляет собой число, а другое – строку, то строка преобразуется в число и выполняется сравнение с преобразованным значением.
  - Если какое-либо значение равно `true`, оно преобразуется в 1 и сравнение выполняется снова. Если какое-либо значение равно `false`, оно преобразуется в 0 и сравнение выполняется снова.
  - Если одно из значений представляет собой объект, а другое – число или строку, объект преобразуется в базовый тип и сравнение выполняется снова. Объект преобразуется в значение базового типа либо с помощью своего метода `toString()`, либо с помощью своего метода `valueOf()`. Встроенные классы базового JavaScript сначала пытаются выполнить преобразование `valueOf()`, а затем `toString()`, кроме класса Date, который всегда выполняет преобразование `toString()`. Объекты, не являющиеся частью базового JavaScript, могут преобразовывать себя в значения базовых типов способом, определенным в их реализации.
  - Любые другие комбинации значений не являются равными.

В качестве примера проверки на равенство рассмотрим сравнение:

```
"1" == true
```

Результат этого выражения равен `true`, то есть эти по-разному выглядящие значения фактически равны. Логическое значение `true` преобразуется в число 1, и сравнение выполняется снова. Затем строка `"1"` преобразуется в число 1. Поскольку оба числа теперь совпадают, оператор сравнения возвращает `true`.

Когда оператор равенства в JavaScript 1.1 пытался преобразовать строку в число и терпел неудачу, он выводил сообщение об ошибке, говорившее, что строка не может быть преобразована, а не преобразовывал строку в `NaN` с возвращением `false` в качестве результата сравнения. Эта ошибка была исправлена в JavaScript 1.2.

5.4.1.1. Равенство и неравенство в Netscape

Оператор `==` всегда ведет себя так, как описано выше, а оператор `!=` всегда ведет себя так, как описано в следующем разделе, но за одним исключением. В клиентском JavaScript в Netscape 4 и более поздних броузерах атрибут `language` тега `<script>` имеет значение «JavaScript 1.2», и оператор равенства ведет себя так же, как оператор идентичности, а оператор неравенства – как оператор неидентичности. Чтобы избежать этой несовместимости, никогда не используйте атрибут `language="JavaScript1.2"` для вставки своего клиентского кода JavaScript. Полный список подобных несовместимостей Java-

Script 1.2 можно найти в разделе 11.6 «Несовместимости Netscape JavaScript 1.2».

5.4.2. Неравенство (!=) и неидентичность (!==)

Операторы `!=` и `!==` выполняют проверки, в точности противоположные операторам `==` и `===`. Оператор `!=` возвращает `false`, если два значения равны друг другу, и `true` в противном случае. Оператор неидентичности `!==` возвращает `false`, если два значения идентичны друг другу, и `true` – в противном случае. Заметьте, что этот оператор стандартизован в ECMAScript v3 и реализован в JavaScript 1.3 и более поздних версиях.

Как мы увидим позднее, оператор `!` осуществляет операцию логического НЕ. Благодаря этому легче запомнить, что `!=` обозначает «не равно», а `!==` – «не идентично». Подробности определения равенства и идентичности для разных типов данных рассмотрены в предыдущем разделе.

5.5. Операторы отношения

В этом разделе описаны операторы отношения в JavaScript. Это операторы, проверяющие отношение (такое как «меньше» или «является свойством») между двумя значениями и возвращающие `true` или `false`, в зависимости от того, существует ли это отношение. Как мы увидим в главе 6, они чаще всего применяются в инструкциях `if` и циклах `while` для управления ходом исполнения программы.

5.5.1. Операторы сравнения

Из всех типов операторов отношения чаще всего используются операторы сравнения – для определения относительного порядка двух величин. Вот список операторов сравнения:

Меньше (<)

Результат оператора `<` равен `true`, если первый операнд меньше, чем второй операнд; в противном случае он равен `false`.

Больше (>)

Результат оператора `>` равен `true`, если его первый операнд больше, чем второй операнд; в противном случае он равен `false`.

Меньше или равно (<=)

Результатом оператора `<=` является `true`, если первый операнд меньше или равен второму операнду; в противном случае результат равен `false`.

Больше или равно (>=)

Результат оператора `>=` равен `true`, если его первый операнд больше второго или равен ему; в противном случае он равен `false`.

Эти операторы позволяют сравнивать операнды любого типа. Однако сравнение может выполняться только для чисел и строк, поэтому операнды, не являющиеся числами или строками, преобразуются. Сравнение и преобразование выполняется следующим образом:

- Если оба операнда являются числами, или преобразуются в числа, они сравниваются как числа.
- Если оба операнда являются строками или преобразуются в строки, они сравниваются как строки.
- Если один операнд является строкой или преобразуется в строку, а другой является числом или преобразуется в число, оператор пытается преобразовать строку в число и выполнить численное сравнение. Если строка не представляет собой число, она преобразуется в `NaN`, и результатом сравнения будет `false`. (В JavaScript 1.1 преобразование строки в число не дает `NaN`, а приводит к ошибке.)
- Если объект может быть преобразован как в число, так и в строку, JavaScript выполняет преобразование в число. Это значит, например, что объекты Date сравниваются как числа, и можно сравнить две даты и определить, какая из них более ранняя.
- Если оба операнда не могут быть успешно преобразованы в числа или строки, операторы всегда возвращают `false`.
- Если один из операндов равен или преобразуется в `NaN`, то результатом оператора сравнения является `false`.

Имейте в виду, что строковое сравнение выполняется строго посимвольно, для числовых значений каждого символа из кодировки Unicode. В некоторых случаях стандарт Unicode допускает кодирование эквивалентных строк с применением различных последовательностей символов, но операторы сравнения JavaScript не обнаруживают этих различий в кодировках; предполагается, что все строки представлены в нормализованной форме. Заметьте, в частности, что сравнение строк чувствительно к регистру, и в кодировке Unicode (по крайней мере, для подмножества ASCII) все прописные буквы «меньше» всех строчных букв. Это правило может приводить к непонятным результатам, – например, строка «Zoo» меньше, чем строка «aardvark» оператора `<`.

При сравнении строк более устойчив метод `String.localeCompare()`, который также учитывает региональные определения «алфавитного порядка». Для сравнения без учета регистра необходимо сначала преобразовать строки в нижний или верхний регистр с помощью методов `String.toLowerCase()` или `String.toUpperCase()`.

Операторы `<=` (меньше или равно) и `>=` (больше или равно) определяют «равенство» двух значений не при помощи операторов равенства или идентичности. Оператор «меньше или равно» определяется просто как «не больше», а оператор «больше или равно» – как «не меньше». Единственное исключение

имеет место, когда один из операндов представляет собой NaN (или преобразуется в него); в этом случае все четыре оператора сравнения возвращают false.

5.5.2. Оператор in

Оператор in требует, чтобы левый операнд был строкой или мог быть преобразован в строку. Правым операндом должен быть объект (или массив). Результатом оператора будет true, если левое значение представляет собой имя свойства объекта, указанного справа. Например:

```
var point = { x:1, y:1 };        // Определяем объект
var has_x_coord = "x" in point;  // Равно true
var has_y_coord = "y" in point;  // Равно true
var has_z_coord = "z" in point;  // Равно false; это не трехмерная точка
var ts = "toString" in point;    // Унаследованное свойство; равно true
```

5.5.3. Оператор instanceof

Оператор instanceof требует, чтобы левым операндом был объект, а правым – имя класса объектов. Результатом оператора будет true, если объект, указанный слева, представляет собой экземпляр класса, указанного справа; в противном случае результатом будет false. В главе 8 мы увидим, что в JavaScript классы объектов определяются функцией-конструктором, их инициализировавшим. Следовательно, правый операнд instanceof должен быть именем функции-конструктора. Заметьте, что все объекты представляют собой экземпляры класса Object. Например:

```
var d = new Date();   // Создаем новый объект с помощью конструктора Date()
d instanceof Date;    // Равно true; объект d был создан с помощью функции Date()
d instanceof Object;  // Равно true; все объекты представляют собой экземпляры
                      // класса Object
d instanceof Number;  // Равно false; d не является объектом Number
var a = [1, 2, 3];    // Создаем массив с помощью литерала массива
a instanceof Array;   // Равно true; a - это массив
a instanceof Object;  // Равно true; все массивы представляют собой объекты
a instanceof RegExp;  // Равно false; массивы не являются регулярными выражениями
```

Если левый операнд instanceof не является объектом, или если правый операнд – это объект, не имеющий функции-конструктора, instanceof возвращает false. Но если правый операнд вообще не является объектом, возвращается ошибка времени выполнения.

5.6. Строковые операторы

Как мы узнали из предыдущих разделов, есть несколько операторов, ведущих себя особым образом, когда в качестве операндов выступают строки.

Оператор + выполняет конкатенацию двух строковых операндов. Другими словами, создает новую строку, состоящую из первой строки, за которой следует вторая строка. Так, следующее выражение равно строке «hello there»:

```
"hello" + " " + "there"
```

Следующие инструкции дают в результате строку «22»:

```
a = "2"; b = "2";
c = a + b;
```

Операторы <, <=, > и >= сравнивают две строки и определяют, в каком порядке они следуют друг за другом. Сравнение основано на алфавитном порядке. Однако, как было отмечено выше, этот алфавитный порядок базируется на используемой в JavaScript кодировке символов Unicode. В этой кодировке все прописные буквы идут раньше, чем все строчные буквы (прописные меньше строчных), что может привести к неожиданным результатам.

Операторы равенства == и неравенства != применяются не только к строкам, но, как мы видели, ко всем типам данных, и при работе со строками ничем особенным не выделяются.

Оператор + особенный – он дает приоритет строковым операндам перед числовыми. Как было замечено ранее, если один из операндов оператора + представляет собой строку (или объект), то другой оператор преобразуется в строку (или оба операнда преобразуются в строки) и операторы конкатенируются, а не складываются. С другой стороны, операторы сравнения выполняют строковое сравнение, только если *оба* операнда представляют собой строки. Если только один операнд – строка, то JavaScript пытается преобразовать ее в число. Вот иллюстрация этих правил:

```
1 + 2        // Сложение. Результат равен 3.
"1" + "2"    // Конкатенация. Результат равен "12".
"1" + 2      // Конкатенация; 2 преобразуется в "2". Результат равен "12".
11 < 3       // Численное сравнение. Результат равен false.
"11" < "3"   // Строковое сравнение. Результат равен true.
"11" < 3     // Численное сравнение; "11" преобразуется в 11.
             // Результат равен false.
"one" < 3    // Численное сравнение; "one" преобразуется в NaN.
             // Результат равен false.
             // В JavaScript 1.1 это приводит к ошибке, а не к NaN.
```

И наконец, важно заметить, что когда оператор + применяется к строкам и числам, он может быть неассоциативным. Другими словами, результат может зависеть от порядка, в котором выполняются операции. Это можно видеть на следующих примерах:

```
s = 1 + 2 + " blind mice";   // Равно "3 blind mice"
t = "blind mice: " + 1 + 2;  // Равно "blind mice: 12"
```

Причина этой удивительной разницы в поведении заключается в том, что оператор + работает слева направо, если только скобки не меняют этот порядок. Следовательно, последние два примера эквивалентны следующему:

```
s = (1 + 2) + "blind mice";    // Результат первой операции - число; 2-й - строка
t = ("blind mice: " + 1) + 2;  // Результаты обеих операций - строки
```

5.7. Логические операторы

Логические операторы обычно используются для выполнения операций булевой алгебры. Они часто применяются в сочетании с операторами сравнения для осуществления сложных сравнений с участием нескольких переменных в инструкциях `if`, `while` и `for`.

5.7.1. Логическое И (&&)

При использовании с логическими операндами оператор `&&` выполняет операцию «логическое И» над двумя значениями: он возвращает `true` тогда и только тогда, когда первый *и* второй операнды равны `true`. Если один или оба операнда равны `false`, оператор возвращает `false`.

Реальное поведение этого оператора несколько сложнее. Он начинает с вычисления левого операнда. Если получившееся значение может быть преобразовано в `false` (если левый операнд равен `null`, `0`, `""` или `undefined`), оператор возвращает значение левого выражения. В противном случае оператор вычисляет правый операнд и возвращает значение этого выражения.

Заметьте, что в зависимости от значения левого выражения этот оператор либо вычисляет, либо не вычисляет правое выражение. Иногда встречается код, намеренно использующий эту особенность оператора `&&`. Так, следующие две строки кода JavaScript дают эквивалентные результаты:

```
if (a == b) stop();
(a == b) && stop();
```

Некоторые программисты (особенно работавшие с Perl) считают такой стиль программирования естественным и полезным, но я не рекомендую прибегать к таким приемам. Тот факт, что вычисление правой части не гарантируется, часто является источником ошибок. Рассмотрим следующий код:

```
if ((a == null) && (b++ > 10)) stop();
```

Скорее всего, эта инструкция не делает того, что предполагал программист, так как оператор инкремента в правой части не вычисляется в тех случаях, когда левое выражение равно `false`. Чтобы обойти этот подводный камень, не помещайте выражения, имеющие побочные эффекты (присваивания, инкременты, декременты и вызовы функций), в правую часть оператора `&&`, если только не уверены абсолютно в том, что делаете.

Несмотря на довольно запутанный алгоритм работы этого оператора, проще всего и абсолютно безопасно рассматривать его как оператор булевой алгебры. На самом деле он не возвращает логического значения, но то значение, которое он возвращает, всегда может быть преобразовано в логическое.

5.7.2. Логическое ИЛИ (||)

При использовании с логическими операндами оператор `||` выполняет операцию «логическое ИЛИ» над двумя значениями: он возвращает `true`, если первый *или* второй операнд (или оба операнда) равны `true`. Если оба операнда равны `false`, он возвращает `false`.

Хотя оператор `||` чаще всего применяется просто как оператор «логическое ИЛИ», он, как и оператор `&&`, ведет себя более сложным образом. Его работа начинается с вычисления левого операнда. Если значение этого выражения может быть преобразовано в `true`, возвращается значение левого выражения. В противном случае оператор вычисляет правый операнд и возвращает значение этого выражения.[1]

Как и в операторе `&&`, следует избегать правых операндов, имеющих побочные эффекты, если только вы умышленно не хотите воспользоваться тем обстоятельством, что правое выражение может не вычисляться.

Даже когда оператор `||` применяется с операндами нелогического типа, его все равно можно рассматривать как оператор «логическое ИЛИ», так как возвращаемое им значение, независимо от типа, может быть преобразовано в логическое.

5.7.3. Логическое НЕ (!)

Оператор `!` представляет собой унарный оператор, помещаемый перед одиночным операндом. Оператор инвертирует значение своего операнда. Так, если переменная `a` имеет значение `true` (или представляет собой значение, преобразуемое в `true`), то `!a` имеет значение `false`. И если выражение `p && q` равно `false` (или значению, преобразуемому в `false`), то `!(p && q)` будет равно `true`. Обратите внимание, что можно преобразовать значение любого типа в логическое, применив этот оператор дважды: `!!x`.

5.8. Поразрядные операторы

Несмотря на то что все числа в JavaScript вещественные, поразрядные операторы требуют в качестве операндов целые числа. Они работают с такими операндами с помощью 32-разрядного целого представления, а не эквивалентного представления с плавающей точкой. Четыре из этих операторов выполняют поразрядные операции булевой алгебры, аналогично описан-

[1] В JavaScript 1.0 и 1.1, если левый операнд может быть преобразован в `true`, оператор возвращает `true`, а не непреобразованное значение.

ным выше логическим операторам, но рассматривая каждый бит операнда как отдельное логическое значение. Три других поразрядных оператора применяются для сдвига битов влево и вправо.

В JavaScript 1.0 и 1.1 поразрядные операторы возвращают `NaN`, если операнды не являются целыми числами или слишком велики и не помещаются в 32-разрядное целое. Однако JavaScript 1.2 и ECMAScript просто втискивают операнды в 32-разрядное целое, отбрасываядробную часть операнда и любые биты старше 32-го. Операторы сдвига требуют, чтобы значение правого операнда было целым числом от 0 до 31. После преобразования этого операнда в 32-разрядное целое вышеописанным образом они отбрасывают любые биты старше 5-го, получая число в соответствующем диапазоне.

Те, кто не знаком с двоичными числами и двоичным представлением десятичных целых чисел, могут пропустить операторы, рассмотренные в этом разделе. Назначение этих операторов здесь не описано; они требуются для низкоуровневых манипуляций с двоичными числами и достаточно редко применяются в программировании на JavaScript. Вот список поразрядных операторов:

Поразрядное И (&)

Оператор `&` выполняет операцию «логическое И» над каждым битом своих операндов. Бит результата будет равен 1, только если равны 1 соответствующие биты обоих операндов. То есть, `0x1234 & 0x00FF` равно `0x0034`.

Поразрядное ИЛИ (|)

Оператор `|` выполняет операцию «логическое ИЛИ» над каждым битом своих операндов. Бит результата будет равен 1, если равен 1 соответствующий бит хотя бы в одном операнде. Например, `9 | 10` равно `11`.

Поразрядное исключающее ИЛИ (XOR) (^)

Оператор `^` выполняет логическую операцию «исключающее ИЛИ» над каждым битом своих операндов. Исключающее ИЛИ обозначает, что должен быть истинен либо первый операнд, либо второй, но не оба сразу. Бит результата устанавливается, если соответствующий бит установлен в одном (но не в обоих) из двух операндов. Например, `9 ^ 10` равно `3`.

Поразрядное НЕ (~)

Оператор `~` – это унарный оператор, указываемый перед своим единственным целым аргументом. Он выполняет инверсию всех битов операнда. Из-за способа представления целых со знаком в JavaScript применение оператора `~` к значению эквивалентно изменению его знака и вычитанию 1. Например, `~0x0f` равно `0xfffffff0`, или `-16`.

Сдвиг влево (<<)

Оператор `<<` перемещает все биты в первом операнде влево на количество позиций, указанное во втором операнде, который должен быть целым числом от 0 до 31. Например, в операции `a << 1` первый бит в `a` становится вторым битом, второй бит `a` становится третьим и т. д. Новым первым битом становится ноль, значение 32-го бита теряется. Сдвиг значения влево

на одну позицию эквивалентен умножению на 2, на две позиции – умножению на 4, и т. д. Например, `7 << 1` равно 14.

Сдвиг вправо с сохранением знака (>>)

Оператор `>>` перемещает все биты своего первого операнда вправо на количество позиций, указанное во втором операнде (целое между 0 и 31). Биты, сдвинутые за правый край, теряются. Самый старший бит (32-й) не меняется, чтобы сохранить знак результата. Если первый операнд положителен, старшие биты результата заполняются нулями; если первый операнд отрицателен, старшие биты результата заполняются единицами. Сдвиг значения вправо на одну позицию эквивалентен делению на 2 (с отбрасыванием остатка), а сдвиг вправо на две позиции эквивалентен делению на 4, и т. д. Например, `7 >> 1` равно 3, а `-7 >> 1` равно –4.

Сдвиг вправо с заполнением нулями (>>>)

Оператор `>>>` аналогичен оператору `>>` за исключением того, что при сдвиге старшие разряды заполняются нулями независимо от знака первого операнда. Например, `-1 >> 4` равно –1, а `-1 >>> 4` равно 268 435 455 (0x0fffffff).

5.9. Операторы присваивания

Как мы видели в обсуждении переменных в главе 4, для присваивания значения переменной в JavaScript используется символ `=`. Например:

```
i = 0
```

Можно не рассматривать такую строку JavaScript как выражение, которое имеет результат, но это действительно выражение и формально «=» представляет собой оператор.

Левым операндом оператора = должна быть переменная, элемент массива или свойство объекта. Правым операндом может быть любое значение любого типа. Значением оператора присваивания является значение правого операнда. Побочный эффект оператора = заключается в присваивании значения правого операнда переменной, элементу массива или свойству, указанному слева, так что при последующих обращениях к переменной, элементу массива или свойству будет получено это значение.

Поскольку = представляет собой оператор, его можно включать в более сложные выражения. Так, в одном выражении можно выполнить присваивание и проверку значения:

```
(a = b) == 0
```

При этом следует отчетливо понимать разницу между операторами = и ==!

Если в выражении присутствует несколько операторов присваивания, они вычисляются справа налево. Поэтому можно написать код, присваивающий одно значение нескольким переменным, например:

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

Помните, что каждое выражение присваивания имеет значение, равное значению правой части. Поэтому в коде, приведенном выше, значение первого присваивания (самого правого) становится правой частью второго присваивания (среднего), а это значение становится правой частью последнего (самого левого) присваивания.

5.9.1. Присваивание с операцией

Кроме обычного оператора присваивания (=) JavaScript поддерживает несколько других операторов-сокращений, объединяющих присваивание с некоторой другой операцией. Например, оператор += выполняет сложение и присваивание. Выражение

```
total += sales_tax
```

эквивалентно

```
total = total + sales_tax
```

Как можно было ожидать, оператор += работает и с числами и со строками. Если операнды числовые, он выполняет сложение и присваивание, а если строковые – конкатенацию и присваивание.

Таблица 5.2. Операторы присваивания

Оператор	Пример	Эквивалент
+=	a += b	a = a + b
-=	a -= b	a = a - b
*=	a *= b	a = a * b
/=	a /= b	a = a / b
%=	a %= b	a = a % b
<<=	a <<= b	a = a << b
>>=	a >>= b	a = a >> b
>>>=	a >>>= b	a = a >>> b
&=	a &= b	a = a & b
|=	a |= b	a = a | b
^=	a ^= b	a = a ^ b

Имеются подобные операторы -=, *=, &= и прочие. В табл. 5.2 приведен их полный список. В большинстве случаев выражение:

```
a op = b
```

где *op* – это оператор, эквивалентно выражению

```
a = a op b
```

Эти выражения отличаются, только если a содержит операции, имеющие побочные эффекты, такие как вызов функции или оператор инкремента.

5.10. Прочие операторы

JavaScript поддерживает еще несколько операторов, которые описаны в следующих разделах.

5.10.1. Условный оператор (?:)

Условный оператор – это единственный тернарный оператор (с тремя операндами) в JavaScript и иногда он так и называется – «тернарный оператор». Оператор иногда записывается как `?:`, хотя в коде он выглядит по-другому. Этот оператор имеет три операнда, первый идет перед `?`, второй – между `?` и `:`, а третий – после `:`. Используется он следующим образом:

```
x > 0 ? x*y : -x*y
```

Первый операнд условного оператора должен быть логическим значением (или преобразовываться в логическое значение) – обычно это результат выражения сравнения. Второй и третий операнды могут быть любыми значениями. Значение, возвращаемое условным оператором, зависит от логического значения первого операнда. Если этот операнд равен `true`, то условное выражение принимает значение второго операнда. Если первый операнд равен `false`, то условное выражение принимает значение третьего операнда.

Тот же результат можно получить с помощью инструкции `if`, но оператор `?:` часто оказывается удобным сокращением. Вот типичный пример, в котором проверяется, определена ли переменная, и если да, то берется ее значение, а если нет, то значение по умолчанию:

```
greeting = "hello " + (username != null ? username : "there");
```

Эта запись эквивалентна следующей конструкции `if`, но более компактна:

```
greeting = "hello ";
if (username != null)
    greeting += username;
else
    greeting += "there";
```

5.10.2. Оператор typeof

Унарный оператор `typeof` помещается перед единственным операндом, который может иметь любой тип. Его значение представляет собой строку, указывающую тип данных операнда.

Результатом оператора `typeof` будет строка «number», «string» или «boolean», если его операндом является число, строка или логическое значение соответственно. Для объектов, массивов и (как ни странно) `null` результатом будет «object». Для операндов-функций результатом будет «function», а для неопределенного операнда – «undefined».

Значение оператора typeof равно «object», когда операнд представляет собой объект-обертку Number, String или Boolean. Оно также равно «object» для объектов Date и RegExp. Для объектов, не являющихся частью базового JavaScript, а предоставляемых контекстом, в который встроен JavaScript, возвращаемое оператором typeof значение зависит от реализации. Однако в клиентском JavaScript typeof обычно равен «object» для всех клиентских объектов – так же, как и для всех базовых объектов.

Оператор typeof может применяться, например, в таких выражениях:

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

Операнд typeof можно заключить в скобки, благодаря чему typeof выглядит как имя функции, а не как ключевое слово или оператор:

```
typeof(i)
```

Для всех объектных типов и типов массивов результатом оператора typeof является «object», поэтому он может быть полезен только для того, чтобы отличить объекты от базовых типов. Для того чтобы отличить один объектный тип от другого, следует обратиться к другим приемам, таким как оператор instanceof или свойство constructor (см. описание Object.constructor в справочном разделе, посвященном базовому языку).

Оператор typeof определен в спецификации ECMAScript v1 и реализован в JavaScript 1.1 и более поздних версиях.

5.10.3. Оператор создания объекта (new)

Оператор new создает новый объект и вызывает функцию-конструктор для его инициализации. Это унарный оператор, указываемый перед вызовом конструктора и имеющий следующий синтаксис:

```
new конструктор(аргументы)
```

Здесь *конструктор* – это выражение, результатом которого является функция-конструктор, и за ним должны следовать ноль или более аргументов, разделенных запятыми и заключенных в круглые скобки. Как особый случай, и только для оператора new, JavaScript упрощает грамматику, допуская отсутствие скобок, если у функции нет аргументов. Вот несколько примеров использования оператора new:

```
o = new Object;  // Здесь необязательные скобки опущены
d = new Date();  // Возвращает объект Date, содержащий текущее время
c = new Rectangle(3.0, 4.0, 1.5, 2.75);  // Создает объект класса Rectangle
obj[i] = new constructors[i]();
```

Оператор new сначала создает новый объект с неопределенными свойствами; затем он вызывает заданную функцию-конструктор, передавая ей указанные аргументы, а также только что созданный объект в качестве значения

ключевого слова this. С помощью этого слова функция-конструктор может инициализировать новый объект любым необходимым образом. В главе 8 оператор new, ключевое слово this и функции-конструкторы рассмотрены более подробно.

Оператор new может также применяться для создания массивов с помощью синтаксиса new Array(). Подробнее о создании объектов и массивов и работе с ними мы поговорм в главах 8 и 9.

5.10.4. Оператор delete

Унарный оператор delete выполняет попытку удалить свойство объекта, элемент массива или переменную, указанную в его операнде.[1] Он возвращает true, если удаление прошло успешно, и false в противном случае. Не все переменные и свойства могут быть удалены: некоторые встроенные свойства из базового и клиентского JavaScript устойчивы к операции удаления, кроме того, не могут быть удалены переменные, определенные пользователем с помощью инструкции var. Если оператор delete вызывается для несуществующего свойства, он возвращает true. (Как ни странно, стандарт ECMAScript определяет, что оператор delete также возвращает true, если его операнд не является свойством, элементом массива или переменной.) Вот несколько примеров применения этого оператора:

```
var o = {x:1, y:2};  // Определяем переменную; инициализируем ее объектом
delete o.x;          // Удаляем одно из свойств объекта; возвращает true
typeof o.x;          // Свойство не существует; возвращает "undefined"
delete o.x;          // Удаляем несуществующее свойство; возвращает true
delete o;            // Объявленную переменную удалить нельзя; возвращает false
delete 1;            // Нельзя удалить целое; возвращает true
x = 1;               // Неявно объявляем переменную без ключевого слова var
delete x;            // Этот вид переменных можно удалять; возвращает true
x;                   // Ошибка времени выполнения: x не определена
```

Заметьте, что удаленное свойство, переменная или элемент массива не просто устанавливается в undefined. Когда свойство удалено, оно прекращает существование. Эта тема обсуждалась в разделе 4.3.2 «Неопределенные и неинициализированные переменные».

Оператор delete стандартизован в спецификации ECMAScript v1 и реализован в JavaScript 1.2 и более поздних версиях. Обратите внимание, что оператор delete существует в JavaScript 1.0 и 1.1, но в этих версиях языка не выполняет фактического удаления. Вместо этого он присваивает указанному свойству, переменной или элементу массива значение null.

---

[1] Тем, кто программировал на C++, следует обратить внимание, что оператор delete в JavaScript совершенно не похож на оператор delete в C++. В JavaScript освобождение памяти выполняется сборщиком мусора автоматически, и беспокоиться о явном освобождении памяти не надо. Поэтому в операторе delete в стиле C++, удаляющем объекты целиком, нет необходимости.

Важно понимать, что оператор delete влияет только на свойства, а не на объекты, на которые эти свойства ссылаются. Взгляните на следующий код:

```
var my = new Object();     // Создаем объект по имени "my"
my.hire = new Date();      // my.hire ссылается на объект Date
my.fire = my.hire;         // my.fire ссылается на тот же объект
delete my.hire;            // свойство hire удалено; возвращает true
document.write(my.fire);   // Но my.fire продолжает ссылаться на объект Date
```

5.10.5. Оператор void

Унарный оператор void указывается перед своим единственным операндом, тип которого может быть любым. Действие этого оператора необычно: он отбрасывает значение операнда и возвращает undefined. Чаще всего этот оператор применяется в URL javascript: на стороне клиента, где позволяет вычислять выражение ради его побочных действий, не отображая в броузере вычисленное значение.

Например, можно использовать оператор void в HTML-теге так:

```
<a href="javascript:void window.open();">Открыть новое окно</a>
```

Другое применение void — это намеренная генерация значения undefined. Оператор void определяется в ECMAScript v1 и реализуется в JavaScript 1.1. В ECMAScript v3 определяется глобальное свойство undefined, реализуемое в JavaScript 1.5. Однако для обратной совместимости может быть полезно обратиться к выражению вроде void 0, а не к свойству undefined.

5.10.6. Оператор «запятая»

Оператор «запятая» очень прост. Он вычисляет свой левый операнд, вычисляет свой правый операнд и возвращает значение правого операнда, то есть следующая строка

```
i=0, j=1, k=2;
```

возвращает значение 2 и практически эквивалентна записи:

```
i = 0;
j = 1;
k = 2;
```

Этот странный оператор полезен лишь в ограниченных случаях, в основном, когда требуется вычислить несколько независимых выражений с побочными эффектами там, где допускается только одно выражение. На практике оператор «запятая» фактически используется только в сочетании с инструкцией for, которую мы рассмотрим в главе 6.

5.10.7. Операторы доступа к массивам и объектам

Как отмечалось в главе 3, можно обращаться к элементам массива посредством квадратных скобок ([]), а к элементам объекта – посредством точки (.). И [], и . рассматриваются в JavaScript как операторы.

Оператору «точка» в качестве левого операнда требуется объект, а в качестве правого – идентификатор (имя свойства). Правый операнд не может быть строкой или переменной, содержащей строку; он должен быть точным именем свойства или метода, без каких-либо кавычек. Вот несколько примеров:

```
document.lastModified
navigator.appName
frames[0].length
document.write("hello world")
```

Если указанное свойство в объекте отсутствует, JavaScript не генерирует ошибку, а возвращает в качестве значения выражения undefined.

Большинство операторов допускают произвольные выражения для всех своих операндов, если только тип операнда в данном случае допустим. Оператор «точка» представляет собой исключение: правый операнд должен быть идентификатором. Ничего другого не допускается.

Оператор [] обеспечивает доступ к элементам массива. Он также обеспечивает доступ к свойствам объекта без ограничений, накладываемых на правый операнд оператора «точка». Если первый операнд (указанный перед левой скобкой) ссылается на массив, то второй операнд (указанный между скобками) должен быть выражением, имеющим целое значение. Например:

```
frames[1]
document.forms[i + j]
document.forms[i].elements[j++]
```

Если первый операнд оператора [] представляет собой ссылку на объект, то второй должен быть выражением, результатом которого является строка, соответствующая имени свойства объекта. Заметьте, что в этом случае второй операнд представляет собой строку, а не идентификатор. Она может быть либо константой, заключенной в кавычки, либо переменной или выражением, ссылающимся на строку. Например:

```
document["lastModified"]
frames[0]['length']
data["val" + i]
```

Оператор [] обычно применяется для доступа к элементам массива. Для доступа к свойствам объекта он менее удобен, чем оператор «точка», так как требует заключения имени свойства в кавычки. Однако когда объект выступает в роли ассоциативного массива, а имена свойств генерируются динамически, оператор «точка» использоваться не может и следует применять оператор []. Чаще всего такая ситуация возникает в случае применения цикла

for/in, рассмотренного в главе 6. Например, следующий код JavaScript использует цикл for/in и оператор [] для вывода имен и значений всех свойств объекта o:

```
for (f in o) {
    document.write('o.' + f + ' = ' + o[f]);
    document.write('<br>');
}
```

5.10.8. Оператор вызова функции

Оператор () предназначен в JavaScript для вызова функций. Этот оператор необычен в том отношении, что не имеет фиксированного количества операндов. Первый операнд – это всегда имя функции или выражение, ссылающееся на функцию. За ним следует левая скобка и любое количество дополнительных операндов, которые могут быть произвольными выражениями, отделенными друг от друга запятыми. За последним операндом следует правая скобка. Оператор () вычисляет все свои операнды и затем вызывает функцию, заданную первым операндом, используя в качестве аргументов оставшиеся операнды.

```
document.close()
Math.sin(x)
alert("Welcome " + name)
Date.UTC(2000, 11, 31, 23, 59, 59)
funcs[i].f(funcs[i].args[0], funcs[i].args[1])
```

6

Инструкции

Как мы видели в предыдущей главе, выражения – это «фразы» JavaScript, а в результате вычисления выражений получаются значения. Входящие в выражения операторы могут иметь побочные эффекты, но обычно выражения ничего не делают. Чтобы что-то произошло, необходимо использовать *инструкции*[1] JavaScript, которые похожи на полноценные предложения обычного языка или команды. В данной главе описано назначение и синтаксис различных инструкций JavaScript. Программа на JavaScript представляет собой набор инструкций, и как только вы познакомитесь с этими инструкциями, вы сможете приступить к написанию программ.

Перед тем как начать разговор об инструкциях JavaScript, вспомним, что в разделе 2.4 «Необязательные точки с запятой» говорилось, что в JavaScript инструкции отделяются друг от друга точками с запятой. Однако если каждая инструкция помещается на отдельной строке, JavaScript допускает их отсутствие. Тем не менее желательно выработать привычку ставить точки с запятой всегда.

6.1. Инструкции-выражения

Простейший вид инструкций в JavaScript – это выражения, имеющие побочные эффекты. Мы видели их в главе 5. Основная категория инструкций-выражений – это инструкции присваивания. Например:

---

[1] Слово «statement» принято переводить на русский язык как «оператор». Однако в контексте большинства современных языков программирования такой перевод приводит к неоднозначности, так как слово «operator» (которое и переводится как «оператор») имеет другое значение. Поэтому здесь и далее в этой книге слово «statement» переводится как «инструкция». – *Примеч. науч. ред.*

```
s = "Привет " + name;
i *= 3;
```

Операторы инкремента и декремента, `++` и `--`, родственны операторам присваивания. Их побочным эффектом является изменение значения переменной, как при выполнении присваивания:

```
counter++;
```

Оператор `delete` имеет важный побочный эффект удаления свойства объекта. Поэтому он почти всегда применяется как инструкция, а не как часть более сложного выражения:

```
delete o.x;
```

Вызовы функций – еще одна большая категория инструкций-выражений. Например:

```
alert("Добро пожаловать, " + name);
window.close();
```

Эти вызовы клиентских функций представляют собой выражения, однако они влияют на веб-броузер, поэтому также являются и инструкциями. Если функция не имеет каких-либо побочных эффектов, нет смысла вызывать ее, если только она не является частью инструкции присваивания. Например, никто не станет просто вычислять косинус и отбрасывать результат:

```
Math.cos(x);
```

Наоборот, надо вычислить значение и присвоить его переменной для дальнейшего использования:

```
cx = Math.cos(x);
```

Снова заметьте, что каждая строка кода в этих примерах завершается точкой с запятой.

6.2. Составные инструкции

В главе 5 мы видели, что объединить несколько выражений в одно можно посредством оператора «запятая». В JavaScript также имеется способ для объединения нескольких инструкций в одну, или в *блок инструкций*. Это делается простым заключением любого количества инструкций в фигурные скобки. Таким образом, следующие строки рассматриваются как одна инструкция, и могут использоваться везде, где JavaScript требует наличия единственной инструкции:

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

Обратите внимание, что хотя блок инструкций действует как одна инструкция, он *не завершается* точкой с запятой. Отдельные инструкции в блоке завершаются точками с запятой, однако сам блок – нет.

Если объединение выражений с помощью оператора «запятая» – это редко используемый прием, то объединение инструкций в блоки кода распространено повсеместно. Как мы увидим в последующих разделах, некоторые инструкции JavaScript сами содержат другие инструкции (так же, как выражения могут содержать другие выражения); эти инструкции называются *составными инструкциями*. Формальный синтаксис JavaScript определяет, что каждая из этих составных инструкций содержит одиночную подынструкцию. Блоки инструкций позволяют помещать любое количество инструкций там, где требуется наличие одной подынструкции.

Исполняя составную инструкцию, интерпретатор JavaScript просто исполняет одну за другой составляющие ее инструкции в том порядке, в котором они записаны. Обычно интерпретатор JavaScript исполняет все инструкции. Однако в некоторых случаях выполнение составной инструкции может быть внезапно прервано. Это происходит, если в составной инструкции содержится инструкция `break`, `continue`, `return` или `throw`, если при выполнении возникает ошибка, или если вызов функции приводит к ошибке или генерации необрабатываемого исключения. Об этих внезапных прерываниях работы мы узнаем больше в последующих разделах.

6.3. if

Инструкция `if` – это базовая управляющая инструкция, позволяющая JavaScript принимать решения или, точнее, исполнять инструкции в зависимости от условий. Инструкция имеет две формы. Первая:

```
if (выражение)
    инструкция
```

В этой форме предложения сначала вычисляется *выражение*. Если полученный результат равен `true` или может быть преобразован в `true`, то исполняется *инструкция*. Если выражение равно `false` или преобразуется в `false`, то *инструкция* не исполняется. Например:

```
if (username == null)       // Если переменная username равна null или undefined,
    username = "John Doe";  // определяем ее
```

или аналогично:

```
// Если переменная username равна null, undefined, 0, "", или NaN,
// она преобразуется в false,
// и эта инструкция присвоит переменной новое значение.
if (!username) username = "John Doe";
```

Несмотря на кажущуюся избыточность, скобки вокруг выражения являются обязательной частью синтаксиса инструкции `if`.

Как было упомянуто в предыдущем разделе, мы всегда можем заменить одиночную инструкцию блоком инструкций. Поэтому инструкция `if` может выглядеть так:

```
if ((address == null) || (address == "")) {
    address = "undefined";
    alert("Пожалуйста, укажите почтовый адрес.");
}
```

Отступы в коде, присутствующие в этих примерах, не обязательны. Дополнительные пробелы и табуляции игнорируются в JavaScript, и поскольку мы ставили после каждой отдельной инструкции точку с запятой, эти примеры могли быть записаны в одну строку. Применение переводов строк и отступов так, как это показано здесь, облегчает чтение и понимание кода.

Вторая форма инструкции `if` вводит конструкцию `else`, исполняемую в тех случаях, когда *выражение* равно `false`. Ее синтаксис:

```
if (выражение)
    инструкция1
else
    инструкция2
```

В этой форме инструкции сначала вычисляется *выражение*, и если оно равно `true`, то исполняется *инструкция1*; в противном случае исполняется *инструкция2*. Например:

```
if (username != null)
    alert("Привет " + username + "\nДобро пожаловать на мою домашнюю страницу.");
else {
    username = prompt("Привет!\n Ваше имя?");
    alert("Привет " + username);
}
```

При наличии вложенных инструкций `if` с блоками `else` требуется некоторая осторожность – необходимо гарантировать, что `else` относится к соответствующей инструкции `if`. Рассмотрим следующие строки:

```
i = j = 1;
k = 2;
if (i == j)
    if (j == k)
        document.write("i равно k");
else
    document.write("i не равно j");    // НЕПРАВИЛЬНО!!
```

В этом примере внутренняя инструкция `if` является одиночной инструкцией внешней инструкции `if`. К сожалению, не ясно (если исключить подсказку, которую дают отступы), к какой инструкции `if` относится блок `else`. И в этом примере отступы выставлены неправильно, ведь интерпретатор JavaScript реально интерпретирует предыдущий пример так:

```
if (i == j) {
    if (j == k)
        document.write("i равно k");
else
    document.write("i не равно j");    // OOPS!
}
```

Правило JavaScript (и большинства других языков программирования) – конструкция `else` является частью ближайшей к ней инструкции `if`. Чтобы сделать этот пример менее двусмысленным и более легким для чтения, понимания, сопровождения и отладки, надо поставить фигурные скобки:

```
if (i == j) {
    if (j == k) {
        document.write("i равно k");
    }
}
else {  // Вот какая разница возникает из-за местоположения фигурных скобок!
    document.write("i не равно j");
}
```

Многие программисты имеют привычку заключать тело инструкций `if` и `else` (а также других составных инструкций, таких как циклы `while`) в фигурные скобки, даже когда тело состоит только из одной инструкции. Последовательное применение этого правила поможет избежать неприятностей, подобных показанной только что.

6.4. else if

Мы видели, что инструкция `if/else` полезна для проверки условия и выполнения одного из двух фрагментов кода в зависимости от результата проверки. Но что если требуется выполнить один из многих фрагментов кода? Один из способов это сделать состоит в применении инструкции `else if`. Формально это не инструкция JavaScript, а лишь распространенный стиль программирования, состоящий в применении повторяющихся инструкций `if/else`:

```
if (n == 1) {
    // Исполняем блок кода #1
}
else if (n == 2) {
    // Исполняем блок кода #2
}
else if (n == 3) {
    // Исполняем блок кода #3
}
else {
    // Если все остальные условия else не выполняются, исполняем блок #4
}
```

В этом коде нет ничего особенного. Это просто последовательность инструкций if, где каждая инструкция if является частью конструкции else предыдущей инструкции. Стиль else if предпочтительнее и понятнее записи в синтаксически эквивалентной форме, полностью показывающей вложенность инструкций:

```
if (n == 1) {
    // Исполняем блок кода #1
}
else {
    if (n == 2) {
        // Исполняем блок кода #2
    }
    else {
        if (n == 3) {
            // Исполняем блок кода #3
        }
        else {
            // Если все остальные условия else не выполняются, исполняем блок #4
        }
    }
}
```

6.5. switch

Инструкция if создает ветвление в потоке выполнения программы. Многопозиционное ветвление можно реализовать посредством нескольких инструкций if, как показано в предыдущем разделе. Однако это не всегда наилучшее решение, особенно если все ветки зависят от значения одной переменной. В этом случае расточительно повторно проверять значение одной и той же переменной в нескольких инструкциях if.

Инструкция switch (реализованная в JavaScript .2 и стандартизованная в ECMAScript v3) работает именно в такой ситуации, и делает это более эффективно, чем повторяющиеся инструкции if. Инструкция switch в JavaScript очень похожа на инструкцию switch в Java или C. За инструкцией switch следует выражение и блок кода – почти так же, как в инструкции if:

```
switch(выражение) {
    инструкции
}
```

Однако полный синтаксис инструкции switch более сложен, чем показано здесь. Различные места в блоке кода помечены ключевым словом case, за которым следует значение и символ двоеточия. Когда выполняется инструкция switch, она вычисляет значение *выражения*, а затем ищет метку case, соответствующую этому значению. Если метка найдена, исполняется блок кода, начиная с первой инструкции, следующей за меткой case. Если метка case с соответствующим значением не найдена, исполнение начинается с первой ин-

струкции, следующей за специальной меткой `default:`. Если метки `default:` нет, блок кода пропускается целиком.

Работу инструкции `switch` сложно объяснить на словах, поэтому приведем пример. Следующая инструкция `switch` эквивалентна повторяющимся инструкциям `if/else`, показанным в предыдущем разделе:

```
switch(n) {
  case 1:                       // Запускается, если n == 1
    // Исполняем блок кода #1.
    break;                      // Здесь останавливаемся
  case 2:                       // Запускается, если n == 2
    // Исполняем блок кода #2.
    break;                      // Здесь останавливаемся
  case 3:                       // Запускается, если n == 3
    // Исполняем блок кода #3.
    break;                      // Здесь останавливаемся
  default:                      // Если все остальное не подходит...
    // Исполняем блок кода #4.
    break;                      // Здесь останавливаемся
}
```

Обратите внимание на ключевое слово `break` в конце каждого `case`. Инструкция `break`, описываемая дальше в этой главе, приводит к передаче управления в конец инструкции `switch` или цикла. Конструкции `case` в инструкции `switch` задают только *начальную точку* выполняемого кода, но не задают никаких конечных точек. В случае отсутствия инструкций `break`, `switch` начинает исполнение блока кода с метки `case`, соответствующей значению *выражения*, и продолжает исполнение до тех пор, пока не дойдет до конца блока. В редких случаях это полезно для написания кода, который переходит от одной метки `case` к следующей, но в 99% случаев следует аккуратно завершать каждый `case` инструкцией `break`. (При использовании `switch` внутри функции можно помещать вместо `break` инструкцию `return`. Обе эти инструкции служат для завершения работы инструкции `switch` и предотвращения перехода к следующему `case`.)

Вот более реальный пример применения инструкции `switch`; он преобразует значение в строку способом, зависящим от типа значения:

```
function convert(x) {
    switch(typeof x) {
      case 'number':            // Преобразуем число в шестнадцатеричное целое
        return x.toString(16);
      case 'string':            // Возвращаем строку, заключенную в кавычки
        return '"' + x + '"';
      case 'boolean':           // Преобразуем в TRUE или FALSE, в верхнем регистре
        return x.toString().toUpperCase();
      default:                  // Любой другой тип преобразуем обычным способом
        return x.toString()
    }
}
```

Заметьте, что в двух предыдущих примерах за ключевыми словами case следовали числа или строковые литералы. Именно так инструкция switch чаще всего используется на практике, но стандарт ECMAScript v3 допускает указание после case произвольного выражения.[1] Например:

```
case 60*60*24:
case Math.PI:
case n+1:
case a[0]:
```

Инструкция switch сначала вычисляет выражение после ключевого слова switch, а затем выражения case в том порядке, в котором они указаны, пока не будет найдено совпадающее значение.[2] Факт совпадения определяется в соответствии с оператором идентичности ===, а не оператором равенства ==, поэтому выражения должны совпадать без какого-либо преобразования типов.

Заметьте, что использование выражений case, содержащих побочные эффекты, такие как вызовы функций и присваивания, не является хорошей практикой программирования, так как при каждом исполнении инструкции switch вычисляются не все выражения case. Когда побочные эффекты возникают лишь в некоторых случаях, трудно понять и предсказать поведение программы. Безопаснее всего ограничиваться в выражениях case константными выражениями.

Как объяснялось ранее, если ни одно из выражений case не соответствует выражению switch, инструкция switch начинает выполнение с инструкции с меткой default:. Если метка default: отсутствует, инструкция switch полностью пропускает свое тело. Обратите внимание, что в предыдущих примерах метка default: указана в конце тела инструкции switch, после всех меток case. Это логичное и обычное место для нее, но на самом деле она может присутствовать в любом месте внутри инструкции switch.

Инструкция switch, реализованая в JavaScript 1.2, не полностью соответствует спецификации ECMAScript. В JavaScript 1.2 выражения case должны быть литералами или константами, вычисляемыми во время компиляции, не содержащими каких-либо переменных или вызовов методов. Кроме того, хотя ECMAScript допускает, чтобы тип выражений switch и case был любым, в JavaScript 1.2 и 1.3 требуется, чтобы результатами выражений были элементарные числовые, строковые или логические значения.

---

[1] Это существенное отличие инструкции switch в JavaScript от инструкции switch в C, C++ и Java. В этих языках выражения case должны быть константами, вычисляемыми при компиляции, иметь тип integer или другой целочисленный тип, и этот тип должен совпадать для всех констант.

[2] Это значит, что инструкция switch в JavaScript менее эффективна, чем инструкция switch в C, C+ и Java. Выражения case в этих языках представляют собой константы, вычисляемые на этапе компиляции, поэтому они никогда не вычисляются во время выполнения, как в JavaScript. Кроме того, поскольку выражения case являются в C, C++ и Java целыми, инструкция switch часто может быть реализована с использованием высокоэффективной таблицы переходов.

6.6. while

Так же как инструкция `if` является базовой управляющей инструкцией, позволяющей JavaScript принимать решения, инструкция `while` — это базовая инструкция, позволяющая JavaScript выполнять повторяющиеся действия. Она имеет следующий синтаксис:

```
while (выражение)
    инструкция
```

Инструкция `while` начинает работу с вычисления *выражения*. Если оно равно `false`, JavaScript переходит к следующей инструкции программы, а если `true`, то исполняется *инструкция*, образующая тело цикла, и *выражение* вычисляется снова. Опять, если значение равно `false`, JavaScript переходит к следующей инструкции программы, в противном случае он исполняет *инструкцию* снова. Цикл продолжается, пока *выражение* не станет равно `false`, тогда инструкция `while` завершит работу, и JavaScript пойдет дальше. Заметьте, что с помощью синтаксиса `while(true)` можно создать бесконечный цикл.

Обычно не требуется, чтобы JavaScript снова и снова выполнял одну и ту же операцию. Почти в каждом цикле с каждой *итерацией* цикла одна или несколько переменных изменяют свои значения. Поскольку переменная меняется, действия, которые выполняет *инструкция*, при каждом выполнении тела цикла могут отличаться. Кроме того, если изменяемая переменная или переменные присутствуют в *выражении*, значение выражения может меняться при каждом выполнении цикла. Это важно, так как в противном случае выражение, значение которого было равно `true`, никогда не изменится и цикл никогда не завершится! Вот пример цикла `while`:

```
var count = 0;
while (count < 10) {
    document.write(count + "<br>");
    count++;
}
```

Как видите, в начале примера переменной `count` присваивается значение 0, а затем ее значение увеличивается каждый раз, когда выполняется тело цикла. После того как цикл будет выполнен 10 раз, выражение становится равным `false` (то есть переменная `count` уже не меньше 10), инструкция `while` завершается, а JavaScript может перейти к следующей инструкции программы. Большинство циклов имеют переменные-счетчики, аналогичные `count`. Чаще всего в качестве счетчиков цикла выступают переменные с именами `i`,`j` и `k`, хотя для того чтобы сделать код более понятным, следует давать счетчикам более наглядные имена.

6.7. do/while

Цикл do/while во многом похож на цикл while, за исключением того, что выражение цикла проверяется в конце цикла, а не в начале. Это значит, что тело цикла всегда исполняется хотя бы один раз. Синтаксис этого предложения таков:

```
do
    инструкция
while (выражение);
```

Инструкция do/while реализована в JavaScript 1.2 и более поздних версиях и стандартизована в ECMAScript v3.

Цикл do/while используется реже, чем родственный ему цикл while. Дело в том, что на практике ситуация, когда требуется хотя бы однократное исполнение цикла, несколько необычна. Например:

```
function printArray(a) {
    if (a.length == 0)
        document.write("Пустой массив");
    else {
        var i = 0;
        do {
          document.write(a[i] + "<br>");
        } while (++i < a.length);
    }
}
```

Есть пара отличий между циклом do/while и обычным циклом while. Во-первых, цикл do требует как ключевого слова do (для отметки начала цикла), так и ключевого слова while (для отметки конца цикла и указания условия). Кроме того, в отличие от цикла while, цикл do завершается точкой с запятой. Причина в том, что цикл do завершается условием цикла, а не просто фигурной скобкой, отмечающей конец тела цикла.

В JavaScript 1.2 имеется ошибка в поведении инструкции continue (см. раздел 6.12 «continue»), когда она помещается внутрь цикла do/while. По этой причине следует избегать применения continue в инструкциях do/while в JavaScript 1.2.

6.8. for

Цикл, начинающийся с инструкции for, часто оказывается более удобным, чем while. Инструкция for использует шаблон, общий для большинства циклов (в том числе приведенного выше примера цикла while). Большинство циклов имеют некоторую переменную-счетчик. Эта переменная инициализируется перед началом цикла и проверяется в *выражении*, вычисляемом перед каждой итерацией цикла. И наконец, переменная-счетчик инкременти-

руется или изменяется каким-либо другим образом в конце тела цикла, непосредственно перед повторным вычислением *выражения*.

Инициализация, проверка и обновление – это три ключевых операции, выполняемых с переменной цикла; инструкция `for` делает эти три шага явной частью синтаксиса цикла. Это особенно облегчает понимание действий, выполняемых данным циклом `for`, и предотвращает такие ошибки, как пропуск инициализации или инкрементирования переменной цикла. Синтаксис цикла `for`:

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

Проще всего объяснить работу цикла `for`, показав эквивалентный ему цикл `while`:[1]

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

Другими словами, выражение *инициализация* вычисляется один раз, перед началом цикла. Это выражение, как правило, является выражением с побочными эффектами (обычно присваиванием), так как от него должна быть какая-то польза. JavaScript также допускает, чтобы выражение *инициализация* было инструкцией объявления переменной `var`, поэтому можно одновременно объявить и проинициализировать счетчик цикла. Выражение *проверка* вычисляется перед каждой итерацией и определяет, будет ли выполняться тело цикла. Если выражение *проверка* равно `true`, выполняется *инструкция*, являющаяся телом цикла. В конце цикла вычисляется выражение *инкремент*. И это выражение, чтобы приносить пользу, должно быть выражением с побочными эффектами. Обычно это либо выражение присваивания, либо выражение, использующее операторы `++` или `--`.

Пример цикла `while` из предыдущего раздела, выводящий числа от 0 до 9, может быть переписан в виде следующего цикла `for`:

```
for(var count = 0; count < 10; count++)
    document.write(count + "<br>");
```

Обратите внимание, что этот синтаксис помещает всю важную информацию о переменной цикла в одну строку, делая работу цикла более понятной. Заметьте также, что помещение выражения *инкремент* в инструкцию `for` само по себе упрощает тело цикла до одной инструкции; нам даже не потребовалось ставить фигурные скобки для формирования блока инструкций.

---

[1] Как мы увидим при рассмотрении инструкции `continue`, этот цикл `while` не является точным эквивалентом цикла `for`.

Конечно, циклы могут быть значительно более сложными, чем в этих простых примерах, и иногда на каждой итерации цикла изменяется несколько переменных. Эта ситуация – единственный случай в JavaScript, когда часто применяется оператор «запятая» – он позволяет объединить несколько выражений инициализации и инкрементирования в одно выражение, подходящее для использования в цикле `for`. Например:

```
for(i = 0, j = 10; i < 10; i++, j--)
    sum += i * j;
```

6.9. for/in

Ключевое слово `for` в JavaScript существует в двух ипостасях. Мы только что видели его в цикле `for`. Оно также используется в инструкции `for/in`. Эта инструкция – несколько иной вид цикла, имеющего следующий синтаксис:

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

Здесь *переменная* должна быть либо именем переменной, либо инструкцией `var`, объявляющей переменную, либо элементом массива, либо свойством объекта (то есть должна быть чем-то, подходящим в качестве левой части выражения присваивания). Параметр *объект* – это имя объекта или выражение, результатом которого является объект. И как обычно, *инструкция* – это инструкция или блок инструкций, образующий тело цикла.

Элементы массива можно перебирать простым увеличением индексной переменной при каждом исполнении тела цикла `while` или `for`. Инструкция `for/in` предоставляет способ перебора всех свойств объекта. Тело цикла `for/in` исполняется единожды для каждого свойства *объекта*. Перед исполнением тела цикла имя одного из свойств объекта присваивается *переменной* в виде строки. В теле цикла эту переменную можно использовать для получения значения свойства объекта с помощью оператора `[]`. Например, следующий цикл `for/in` печатает имя и значение всех свойств объекта:

```
for (var prop in my_object) {
    document.write("имя: " + prop + "; значение: " + my_object[prop], "<br>");
}
```

Обратите внимание, что *переменной* в цикле `for/in` может быть любое выражение, если только результатом его является нечто, подходящее для левой стороны присваивания. Это выражение вычисляется при каждом вызове тела цикла, то есть каждый раз оно может быть различным. Так, скопировать имена всех свойств объекта в массив можно посредством следующего кода:

```
var o = {x:1, y:2, z:3};
var a = new Array();
var i = 0;
for(a[i++] in o) /* пустое тело цикла */;
```

Массивы JavaScript – это просто специальный тип объектов. Следовательно, цикл `for/in` может применяться для перебора элементов массива так же, как для свойств объекта. Например, предыдущий блок кода при замене строки на приведенную ниже перечисляет «свойства» массива 0, 1 и 2:

```
for(i in a) alert(i);
```

Цикл `for/in` не задает порядка, в котором свойства объекта присваиваются переменной. Нельзя заранее узнать, каким будет этот порядок, и в различных реализациях и версиях JavaScript поведение может быть разным. Если тело цикла `for/in` удалит свойство, которое еще не было перечислено, это свойство уже перечислено не будет. Если тело цикла определяет новые свойства, то будут или нет перечислены эти свойства, зависит от реализации.

Цикл `for/in` на самом деле не перебирает все свойства всех объектов. Так же как некоторые свойства объектов помечены, как доступные только для чтения или постоянные (не удаляемые), некоторые свойства помечены как не перечисляемые. Эти свойства не перечисляются циклом `for/in`. Если все свойства, определенные пользователем, перечисляются, то многие встроенные свойства, включая все встроенные методы, не перечисляются. Как мы увидим в главе 8, объекты могут наследовать свойства от других объектов. Наследованные свойства, которые определены пользователем, также перечисляются циклом `for/in`.

6.10. Метки

Метки `case` и `default:` в сочетании с инструкцией `switch` – это особый вариант более общего случая. В JavaScript 1.2 любая инструкция может быть помечена указанным перед ней именем идентификатора и двоеточием:

```
идентификатор: инструкция
```

Здесь *идентификатор* может быть любым допустимым идентификатором JavaScript, не являющимся зарезервированным словом. Имена меток отделены от имен переменных и функций, поэтому программист не должен беспокоиться о конфликте имен, если имя метки совпадает с именем переменной или функции. Вот пример инструкции `while` с меткой:

```
parser:
  while(token != null) {
      // здесь код опущен
}
```

Пометив инструкцию, мы даем ей имя, по которому на нее можно ссылаться из любого места программы. Пометить можно любую инструкцию, хотя обычно помечаются только циклы `while`, `do/while`, `for` и `for/in`. Дав циклу имя, можно посредством инструкции `break` и `continue` выходить из цикла или из отдельной итерации цикла.

6.11. break

Инструкция break приводит к немедленному выходу из самого внутреннего цикла или инструкции switch. Синтаксис ее прост:

```
break;
```

Инструкция break приводит к выходу из цикла или инструкции switch, поэтому такая форма break допустима только внутри этих инструкций.

ECMAScript v3 и JavaScript 1.2 допускают указание имени метки за ключевым словом break:

```
break: имя_метки;
```

Заметьте, что *имя_метки* – это просто идентификатор; за ним не указывается двоеточие, как в случае определения метки нструкции.

Когда break используется с меткой, происходит переход в конец или прекращение выполнения именованной инструкции, которая может быть любой инструкцией, внешней по отношению к break. Именованная инструкция не обязана быть циклом или switch; инструкция break, использованная с меткой, даже не обязана находиться внутри цикла или инструкции switch. Единственное ограничение на метку, указанную в инструкции break, – она должна быть именем *внешней* по отношению к break инструкции. Метка может быть, например, именем инструкции if или даже блока инструкций, заключенных в фигурные скобки только для присвоения метки этому блоку.

Как обсуждалось в главе 2, между ключевым словом break и именем метки перевод строки не допускается. Дело в том, что JavaScript автоматически вставляет пропущенные точки с запятой. Если разбить строку кода между ключевым словом break и следующей за ним меткой, JavaScript предполагает, что вы имели в виду простую форму этой инструкции без метки, и добавляет точку с запятой.

Мы уже видели примеры инструкции break, помещенной в инструкцию switch. В циклах она обычно применяется для преждевременного выхода в тех случаях, когда по какой-либо причине уже нет необходимости доводить выполнение цикла до конца. Когда в цикле имеются сложные условия выхода, часто проще реализовать некоторые из этих условий с помощью инструкции break, а не пытаться включить их все в одно выражение цикла.

Следующий код выполняет поиск определенного значения среди элементов массива. Цикл прерывается естественным образом, когда доходит до конца массива; если искомое значение найдено, он прерывается с помощью инструкции break:

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

Форма инструкции break с меткой требуется только во вложенных циклах или в инструкции switch при необходимости выйти из инструкции, не являющейся самой внутренней.

Следующий пример показывает помеченные циклы for и инструкции break с метками. Проверьте, удастся ли вам понять, каким будет результат работы этого кода:

```
outerloop:
  for(var i = 0; i < 10; i++) {
    innerloop:
      for(var j = 0; j < 10; j++) {
          if (j > 3) break;              // Выход из самого внутреннего цикла
          if (i == 2) break innerloop;   // То же самое
          if (i == 4) break outerloop;   // Выход из внешнего цикла
          document.write("i = " + i + " j = " + j + "<br>");
      }
  }
  document.write("FINAL i = " + i + " j = " + j + "<br>");
```

6.12. continue

Инструкция continue схожа с инструкцией break. Однако вместо выхода из цикла continue запускает новую итерацию цикла. Синтаксис инструкции continue столь же прост, как и у инструкции break:

```
continue;
```

В ECMAScript v3 и JavaScript 1.2 инструкция continue может также использоваться с меткой:

```
continue имя_метки;
```

Инструкция continue, как в форме без метки, так и с меткой, может применяться только в теле циклов while, do/while, for или for/in. Использование ее в любых других местах приводит к синтаксической ошибке.

Когда выполняется инструкция continue, текущая итерация цикла прерывается и начинается следующая. Для разных типов циклов это означает разное:

- В цикле while *выражение*, указанное в начале цикла, проверяется снова и, если оно равно true, тело цикла выполняется сначала.
- В цикле do/while исполнение переходит в конец цикла, где, перед повторным исполнением цикла, снова проверяется условие. Однако заметьте, что в JavaScript 1.2 имеется ошибка, приводящая к тому, что инструкция continue передает управление в начало цикла do/while без проверки условия цикла. Поэтому следует избегать применения инструкции continue в цикле do/while. Однако это препятствие легко обойти, ведь всегда можно заменить цикл do/while эквивалентным циклом while.

- В цикле `for` вычисляется выражение *инкремент* и снова проверяется выражение *проверка* – чтобы определить, следует ли выполнять следующую итерацию.
- В цикле `for/in` цикл начинается заново с присвоением указанной переменной имени следующего свойства.

Обратите внимание на различие в поведении инструкции `continue` в циклах `while` и `for` – цикл `while` возвращается непосредственно к своему условию, а цикл `for` сначала вычисляет выражение *инкремент*, а затем возвращается к условию. Ранее, в обсуждении цикла `for`, я объяснял поведение цикла `for` в терминах эквивалентного цикла `while`. Поскольку инструкция `continue` ведет себя в этих двух циклах по-разному, невозможно точно имитировать цикл `for` с помощью цикла `while`.

В следующем примере показано применение инструкции `continue` без метки для выхода из текущей итерации цикла при возникновении ошибки:

```
for(i = 0; i < data.length; i++) {
    if (data[i] == null)
        continue;  // Продолжение с неопределенными данными невозможно
    total += data[i];
}
```

Инструкция `continue`, как и `break`, может применяться во вложенных циклах в форме, включающей метку, и тогда заново запускаемый цикл – это не обязательно цикл, непосредственно содержащий инструкцию `continue`. Кроме того, как для инструкции `break`, переводы строк между `continue` и именем метки не допускаются.

6.13. var

Инструкция `var` позволяет явно объявить одну или несколько переменных. Синтаксис этой инструкции:

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

За ключевым словом `var` следует список объявляемых переменных через запятую; каждая переменная в списке может иметь специальное выражение-инициализатор, указывающее ее начальное значение. Например:

```
var i;
var j = 0;
var p, q;
var greeting = "hello" + name;
var x = 2.34, y = Math.cos(0.75), r, theta;
```

Инструкция `var` определяет каждую из перечисленных переменных путем создания свойства с этим именем в объекте вызова функции, в которой она находится, или, если объявление находится не в теле функции, то в глобальном объекте. Свойство или свойства, создаваемые с помощью инструкции

var, не могут быть удалены оператором delete. Заметьте, что помещение инструкции var внутрь инструкции with (см. раздел 6.18 «with») не изменяет ее поведения.

Если в инструкции var начальное значение переменной не указано, то переменная определяется, однако ее начальное значение остается неопределенным (undefined).

Заметьте, что инструкция var может также присутствовать как часть циклов for и for/in. Например:

```
for(var i = 0; i < 10; i++) document.write(i, "<br>");
for(var i = 0, j=10; i < 10; i++,j--) document.write(i*j, "<br>");
for(var i in o) document.write(i, "<br>");
```

Значительно больше информации о переменных JavaScript и их объявлении содержится в главе 4.

6.14. function

Инструкция function определяет функцию JavaScript. Она имеет следующий синтаксис:

```
function имя_функции([арг1 [,арг2 [..., аргn]]]) {
    инструкции
}
```

Здесь *имя_функции* – это имя определяемой функции. Оно должно быть идентификатором, а не строкой или выражением. За именем функции следует заключенный в скобки список имен аргументов, разделенных запятыми. Эти идентификаторы могут использоваться в теле функции для ссылки на значения аргументов, переданных при вызове функции.

Тело функции состоит из произвольного числа инструкций JavaScript, заключенных в фигурные скобки. Эти инструкции не исполняются при определении функции. Они компилируются и связываются с новым объектом функции для исполнения при ее вызове с помощью оператора вызова (). Обратите внимание, что фигурные скобки – это обязательная часть инструкции function. В отличие от блоков инструкций в циклах while и других конструкциях, тело функции требует фигурных скобок, даже если оно состоит только из одной инструкции.

Определение функции создает новый объект функции и сохраняет объект в только что созданном свойстве с именем *имя_функции*. Вот несколько примеров определений функций:

```
function welcome() { alert("Добро пожаловать на мою домашнюю страницу!"); }

function print(msg) {
    document.write(msg, "<br>");
}
```

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

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

Определения функций обычно находятся в коде JavaScript верхнего уровня. Они также могут быть вложенными в определения других функций, но только на «верхнем уровне», то есть определения функции не могут находиться внутри инструкций `if`, циклов `while` или любых других конструкций.

Формально `function` не является инструкцией. Инструкции приводят к некоторым динамическим действиям в программе JavaScript, а определения функций описывают статическую структуру программы. Инструкции исполняются во время выполнения, а функции определяются во время анализа или компиляции кода JavaScript – до того как они фактически будут выполнены. Когда синтаксический анализатор JavaScript встречает определение функции, он анализирует и сохраняет (без исполнения) составляющие тело функции инструкции. Затем он определяет свойство (в объекте вызова, если определение функции вложено в другую функцию; в противном случае – в глобальном объекте) с тем же именем, что было указано в определении функции.

Тот факт, что функции определяются на этапе синтаксического анализа, а не во время выполнения, приводит к некоторым удивительным эффектам. Рассмотрим следующий код:

```
alert(f(4));      // Показывает 16. Функция f() может быть вызвана до того, как
                  // она определена.
var f = 0;        // Эта инструкция переписывает содержимое свойства f.
function f(x) {   // Эта "инструкция" определяет функцию f до того, как будут
    return x*x;   // выполнены приведенные выше строки.
}
alert(f);         // Показывает 0. Функция f() была перекрыта переменной f.
```

Такие необычные результаты возникают из-за того, что функция определяется не в то же самое время, что и переменная. К счастью, подобные ситуации возникают не очень часто.

В главе 7 мы узнаем о функциях больше.

6.15. return

Как вы помните, вызов функции с помощью оператора `()` – это выражение. Все выражения имеют значения, и инструкция `return` служит для указания значения, возвращаемого функцией. Это значение становится значением выражения вызова функции. Синтаксис инструкции `return` таков:

```
return выражение;
```

Инструкция `return` может присутствовать только в теле функции. Присутствие ее в любом другом месте является синтаксической ошибкой. Когда выполняется инструкция `return`, вычисляется *выражение* и его значение возвращается в качестве значения функции. Инструкция `return` прекращает исполнение функции, даже если в теле функции остались другие инструкции. Вот как инструкция `return` может применяться для возврата значения:

```
function square(x) { return x*x; }
```

Инструкция `return` может также использоваться без выражения, тогда она прерывает исполнение функции, не возвращая значение. Например:

```
function display_object(obj) {
    // Сначала убедимся в корректности нашего аргумента
    // В случае некорректности пропускаем остаток функции
    if (obj == null) return;
    // Здесь находится оставшаяся часть функции...
}
```

Если в функции выполняется инструкция `return` без выражения, или если она прекращается по причине достижения конца тела функции, значение выражения вызова функции будет неопределенным – `undefined`.

JavaScript вставляет точку с запятой автоматически, поэтому нельзя разделять переводом строки инструкцию `return` и следующее за ней выражение.

6.16. throw

Исключение – это сигнал, указывающий на возникновение какого-либо исключительного обстоятельства или ошибки. *Генерация* (*throw*) исключения – это способ просигнализировать о такой ошибке или исключительном обстоятельстве. *Перехватить* (*catch*) исключение – это значит обработать его: предпринять действия, необходимые или подходящие для восстановления после исключения. В JavaScript исключения генерируются в тех случаях, когда возникает ошибка времени выполнения, и когда программа явно генерирует его с помощью инструкции `throw`. Исключения перехватываются с помощью инструкции `try/catch/finally`, которая будет описана в следующем разделе.[1]

Инструкция `throw` имеет следующий синтаксис:

```
throw выражение;
```

Результатом *выражения* может быть значение любого типа. Однако обычно это объект Error или экземпляр одного из подклассов Error. Также бывает удоб-

[1] Инструкции `throw` и `try/catch/finally` в JavaScript схожи с соответствующими инструкциями в C++ и Java.

но использовать в качестве выражения строку, содержащую сообщение об ошибке или числовое значение, обозначающее некоторый код ошибки.

Вот пример кода, в котором инструкция throw применяется для возбуждения исключительной ситуации:

```
function factorial(x) {
    // Если входной аргумент не является допустимым,
    // возбуждаем исключение!
    if (x < 0) throw new Error("x не может быть отрицательным");
    // В противном случае вычисляем значение и нормальным образом возвращаемся
    // из функции
    for(var f = 1; x > 1; f *= x, x--) /* пусто */ ;
    return f;
}
```

Когда генерируется исключение, интерпретатор JavaScript немедленно прерывает нормальное исполнение программы и переходит к ближайшему обработчику исключений. В обработчиках исключений применяется конструкция catch инструкции try/catch/finally, описание которой приведено в следующем разделе. Если блок кода, в котором возникло исключение, не имеет соответствующей конструкции catch, интерпретатор анализирует следующий внешний блок кода и проверяет, связан ли с ним обработчик исключений. Это продолжается до тех пор, пока обработчик не будет найден. Если исключение генерируется в функции, не содержащей инструкции try/catch/finally, предназначенной для его обработки, то исключение распространяется на код, вызвавший функцию. Так исключения распространяются по лексической структуре методов JavaScript вверх по стеку вызовов. Если обработчик исключения так и не найден, исключение рассматривается как ошибка и о ней сообщается пользователю.

Инструкция throw стандартизована в ECMAScript и реализована в JavaScript 1.4. Класс Error и его подклассы также являются частью стандарта ECMAScript v3, но не реализованы до JavaScript 1.5.

6.17. try/catch/finally

Инструкция try/catch/finally — это механизм обработки исключений в JavaScript. Конструкция try в этой инструкции просто определяет блок кода, в котором исключения будут обрабатываться. За блоком try следует конструкция catch с блоком инструкций, вызываемых, когда где-либо в блоке try возникает исключение. За конструкцией catch следует блок finally, содержащий код зачистки, который гарантированно выполняется независимо от того, что происходит в блоке try. И блок catch и блок finally не являются обязательными, однако после блока try должен обязательно присутствовать хотя бы один из них. Блоки try, catch и finally начинаются и заканчиваются фигурными скобками. Это обязательная часть синтаксиса и она не может быть опущена, даже если между ними содержится только одна инструкция.

Как и инструкция throw, инструкция try/catch/finally стандартизована в ECMAScript v3 и реализована в JavaScript 1.4.

Следующий код иллюстрирует синтаксис и суть инструкции try/catch/finally. Заметьте, в частности, что за ключевым словом catch следует идентификатор в скобках. Этот идентификатор похож на аргумент функции. Он присваивает имя локальной переменной, существующей только в теле блока catch. JavaScript присваивает этой переменной объект исключения или значение, указанное при генерации исключения:

```
try {
  // Обычно этот код без сбоев работает от начала до конца.
  // Но в какой-то момент в нем может генерироваться исключение,
  // либо непосредственно с помощью инструкции throw, либо косвенно, вызовом
  // метода, генерирующего исключение.
}
catch (e) {
  // Инструкции в этом блоке выполняются тогда и только тогда, когда
  // в блоке try генерируется исключение. Эти инструкции могут
  // использовать локальную переменную e, ссылающуюся на объект Error или
  // на другое значение, указанное в инструкции throw. Этот блок может
  // либо каким-либо образом обработать исключение, либо проигнорировать
  // его, делая что-то другое, либо заново сгенерировать исключение
  // с помощью throw.
}
finally {
  // Этот блок содержит инструкции, которые выполняются всегда, независимо
  // от того, что произошло в блоке try. Они выполняются, если блок try
  // прерван:
  //   1) нормальным образом, достигнув конца блока
  //   2) из-за инструкции break, continue, или return
  //   3) с исключением, обработанным приведенным выше блоком catch
  //   4) с неперехваченным исключением, которое продолжает свое
  //      распространение на более высокие уровни
}
```

Вот более реалистичный пример инструкции try/catch. В нем применяются метод factorial(), определенный в предыдущем разделе, и методы клиентского JavaScript prompt() и alert() для ввода и вывода:

```
try {
    // Просим пользователя ввести число
    var n = prompt("Введите положительное число", "");
    // Вычисляем факториал числа, предполагая, что входные данные
    // корректны
    var f = factorial(n);
    // Показываем результат
    alert(n + "! = " + f);
}
catch (ex) {  // Если введенные данные не корректны, мы попадем сюда
    // Сообщаем пользователю об ошибке
```

```
    alert(ex);
}
```

Это пример инструкции try/catch без конструкции finally. Хотя finally не используется так часто, как catch, эта конструкция часто оказывается полезна. Однако ее поведение требует дополнительных объяснений. Блок finally гарантированно исполняется, если исполнялась хотя бы какая-то часть блока try, независимо от того, каким образом завершился код в блоке try. Эта возможность обычно используется для зачистки после кода в предложении try.

В обычной ситуации управление доходит до конца блока try, а затем переходит к блоку finally, который выполняет всю необходимую зачистку. Если управление вышло из блока try из-за инструкций return, continue или break, перед передачей управления в другое место кода исполняется блок finally.

Если в блоке try возникает исключение и имеется соответствующий блок catch для ее обработки, управление сначала передается в блок catch, а затем – в блок finally. Если отсутствует локальный блок catch, то управление сначала передается в блок finally, а затем переходит на ближайший внешний блок catch, который может обработать исключение.

Если сам блок finally передает управление с помощью инструкций return, continue, break или throw, или с помощью вызова метода, генерирующего исключение, незаконченная команда на передачу управления отменяется и выполняется новая. Например, если блок finally генерирует исключение, это исключение заменяет любое исключение, которое было сгенерировано. Если в блоке finally имеется инструкция return, происходит нормальный выход из метода, даже если генерировалось исключение, которое не было обработано.

Инструкции try и finally могут использоваться вместе без конструкции catch. В этом случае блок finally – это просто код зачистки, который будет гарантированно исполнен, независимо от наличия инструкций break, continue или return в блоке try. Например, в следующем коде используется инструкция try/finally, гарантирующая, что счетчик цикла будет инкрементирован в конце каждой итерации, даже если итерация была внезапно прервана инструкцией continue:

```
var i = 0, total = 0;
while(i < a.length) {
    try {
        if ((typeof a[i] != "number") || isNaN(a[i])) // Если это не число,
            continue;   // переходим к следующей итерации цикла.
        total += a[i];  // В противном случае, добавляем число к общей сумме.
    }
    finally {
        i++;  // Всегда увеличиваем i, даже если ранее была
              // инструкция continue.
    }
}
```

6.18. with

В главе 4 мы обсуждали область видимости переменных и цепочку областей видимости – список объектов, в которых выполняется поиск при выполнении разрешения имени переменной. Инструкция `with` применяется для временного изменения цепочки областей видимости. Она имеет следующий синтаксис:

```
with (объект)
    инструкция
```

Эта инструкция добавляет *объект* в начало цепочки областей видимости, исполняет *инструкцию*, а затем восстанавливает цепочку в ее первоначальное состояние.

На практике инструкция `with` помогает значительно сократить объем набираемого текста. В клиентском JavaScript часто работают с глубоко вложенными иерархиями объектов. Например, для доступа к элементам HTML-формы вам, возможно, придется печатать такие выражения:

```
frames[1].document.forms[0].address.value
```

Если надо обратиться к этой форме несколько раз, можно воспользоваться инструкцией `with` для добавления формы в цепочку областей видимости:

```
with(frames[1].document.forms[0]) {
    // Здесь обращаемся к элементам формы непосредственно. Например:
    name.value = "";
    address.value = "";
    email.value = "";
}
```

Это сокращает объем текста – больше не надо указывать `frames[1].document.forms[0]` перед каждым именем свойства. Этот объект представляет собой временную часть цепочки областей видимости и автоматически участвует в поиске, когда JavaScript требуется разрешить такой идентификатор, как `address`.

Несмотря на полезность этой конструкции в некоторых случаях, ее применение осуждается. Код JavaScript, использующий инструкцию `with`, сложен в оптимизации и поэтому может работать медленнее, чем эквивалентный код, написанный без нее. Кроме того, определения функций и инициализации переменных в теле инструкции `with` могут приводить к удивительным и трудным для понимания результатам.[1] По этим причинам не рекомендуется применять инструкцию `with`.

Заметьте, что есть и другие, абсолютно законные способы уменьшения объема набираемого текста. Так, предыдущий пример можно переписать следующим образом:

---

[1] Эти результаты и их причины слишком сложны, чтобы объяснять их здесь.

```
var form = frames[1].document.forms[0];
form.name.value = "";
form.address.value = "";
form.email.value = "";
```

6.19. Пустая инструкция

И наконец, последняя из допустимых в JavaScript инструкций – пустая инструкция. Она выглядит следующим образом:

```
;
```

Выполнение пустой инструкции, очевидно, не имеет никакого эффекта и не производит никаких действий. Можно подумать, что особых причин для ее применения нет, однако изредка пустая инструкция бывает полезной, когда вам требуется создать цикл, имеющий пустое тело. Например:

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

Обратите внимание, что случайное указание точки с запятой после правой круглой скобки циклов `for`, `while` или инструкции `if` может привести к неприятным ошибкам, которые сложно обнаружить. Например, следующий код вряд ли делает то, что предполагал его автор:

```
if ((a == 0) || (b == 0));   // Ой! Эта строка ничего не делает...
    o = null;                // и эта строка всегда будет выполнена.
```

Когда пустая инструкция применяется специально, код желательно снабжать исчерпывающими комментариями. Например:

```
for(i=0; i < a.length; a[i++] = 0) /* Пустое */ ;
```

6.20. Итоговая таблица инструкций JavaScript

В этой главе мы представили все инструкции языка JavaScript. В табл. 6.1 содержится сводка этих инструкций с указанием синтаксиса и назначения каждой из них.

Таблица 6.1. Синтаксис инструкций JavaScript

Инструкция	Синтаксис	Назначение
`break`	`break;` `break` *`имя_метки`*;	Выход из самого внутреннего цикла, инструкции `switch` или инструкции с именем *`имя_метки`*
`case`	`case` *`выражение`*:	Метка для инструкции внутри конструкции `switch`

Инструкция	Синтаксис	Назначение
continue	continue; continue *имя_метки*;	Перезапуск самого внутреннего цикла или цикла, помеченного *меткой*
default	default:	Отметка инструкции по умолчанию внутри инструкции switch
do/while	do *инструкция* while (*выражение*);	Альтернатива циклу while
пустая	;	Не делает ничего
for	for (*инициализация* ; *проверка* ; *инкремент*) *инструкция*	Легкий в использовании цикл
for/in	for (*переменная* in *объект*) *инструкция*	Цикл по свойствам объекта

function	function *имя_функции*([*арг1*[..., *аргn*]])
	{
	инструкции
	}
		Объявление функции

if/else	if (*выражение*) *инструкция1* [else *инструкция2*]	Условное выполнение кода
метка	*идентификатор*: *инструкция*	Присваивание *инструкции* имени *идентификатор*
return	return [*выражение*];	Возврат из функции или задания возвращаемого функцией значения равным *выражению*

switch	switch (*выражение*) {
	инструкции
	}
		Многопозиционное ветвление для инструкций, помеченных метками *case* и *default*

throw	throw *выражение*;	Генерация исключения

try	try {
	инструкции
	}
	catch (*идентификатор*) {
	инструкции
	}
	finally {
	инструкции
	}
		Перехват исключения

Таблица 6.1 (продолжение)

Инструкция	Синтаксис	Назначение
var	var *имя_1* [= *значение_1*] [..., *имя_n* [= *значение_n*]];	Объявление и инициализация переменных
while	while (*выражение*) *инструкция*	Базовая конструкция для цикла
with	with (*объект*) *инструкция*	Расширение цепочки областей видимости. (Не рекомендуется использовать.)

7

Функции

Функции – это важная и сложная часть языка JavaScript. В данной главе функции рассматриваются с разных точек зрения. Сначала мы обсудим функции с точки зрения синтаксиса и расскажем, как они определяются и вызываются. Затем мы рассмотрим их как тип данных, с примерами полезных приемов программирования, возможных, если функции рассматриваются как данные. И наконец, мы рассмотрим вопрос области действия переменных в теле функции и некоторые полезные свойства, доступные выполняемой функции. В том числе обсуждается создание в JavaScript функций, принимающих произвольное количество параметров.

В этой главе мы сосредоточимся на определении и вызове пользовательских функций JavaScript. Важно помнить, что JavaScript поддерживает некоторое количество встроенных функций, таких как `eval()`, `parseInt()` и метод `sort()` класса Array. В клиентском JavaScript определяются другие функции, такие как `document.write()` и `alert()`. Встроенные функции JavaScript применяются точно так же, как и функции, определенные пользователем. О функциях, упомянутых здесь, более подробно можно узнать в справочных разделах этой книги, посвященных базовому и клиентскому JavaScript.

Функции и объекты в JavaScript тесно связаны между собой. По этой причине мы отложим обсуждение некоторых возможностей функций до главы 8.

7.1. Определение и вызов функций

Как мы видели в главе 6, самый распространенный способ определения функции – это инструкция `function`. Она состоит из ключевого слова `function`, за которым следует:

- имя функции;
- заключенный в круглые скобки необязательный список имен параметров, разделенных запятыми;
- инструкции JavaScript, составляющие тело функции, заключенные в фигурные скобки.

В примере 7.1 показаны определения некоторых функций. Хотя эти функции короткие и простые, все они содержат каждый из элементов, которые я только что перечислил. Заметьте, что в функциях может быть определено различное количество аргументов, и что они могут содержать или не содержать инструкцию `return`. Инструкция `return` была описана в главе 6; она прекращает выполнение функции и возвращает значение указанного в ней выражения (если оно есть) вызывающей стороне. Если функция не содержит инструкцию `return`, она просто выполняет все инструкции в своем теле и возвращает неопределенное значение (`undefined`).

Пример 7.1. Определение функций JavaScript

```
// Функция-обертка, иногда ее удобно применять вместо
// document.write().
// В этой функции отсутствует инструкция return, поэтому она не возвращает
// значение.
function print(msg)
{
    Document.write(msg, "<br>");
}
// Функция, вычисляющая и возвращающая расстояние между двумя точками.
function distance(x1, y1, x2, y2)
{
    var dx = x2 - x1;
    var dy = y2 - y1;
    return Math.sqrt(dx*dx + dy*dy);
}
// Рекурсивная функция (вызывающая сама себя), вычисляющая факториалы.
// Вспомните, что x! - это произведение x и всех положительных целых чисел,
// меньших x.
function factorial(x)
{
    if (x <= 1)
        return 1;
    return x * factorial(x-1);
}
```

Будучи один раз определенной, функция может вызываться с помощью оператора `()`, описанного в главе 5. Как вы помните, скобки указываются после имени функции, а необязательный список значений (или выражений) аргументов указывается в скобках через запятую. Функции, определенные в примере 7.1, могут быть вызваны с помощью следующего кода:

```
print("Привет, " + name);
print("Добро пожаловать на мою домашнюю страницу!");
```

```
total_dist = distance(0,0,2,1) + distance(2,1,3,5);
print("Вероятность этого равна: " + factorial(39)/factorial(52));
```

При вызове функции все выражения, указанные между скобками, вычисляются, и полученные значения используются в качестве аргументов функции. Эти значения присваиваются параметрам, имена которых перечислены в определении функции, и функция работает с ними, ссылаясь на эти параметры по указанным именам. Заметьте, что эти переменные-параметры определены, только пока выполняется функция; они не сохраняются после завершения ее работы.

JavaScript – нетипизированный язык, поэтому не требуется указывать тип параметров функций, и JavaScript не проверяет, соответствует ли тип данных требованиям функции. Если тип аргумента важен, вы можете проверить его самостоятельно с помощью оператора `typeof`. Кроме того, JavaScript не проверяет, правильное ли количество параметров передано функции. Если аргументов больше, чем требуется функции, то дополнительные значения просто игнорируются. Если аргументов меньше, то некоторым параметрам присваивается значение `undefined` – что во многих случаях приводит к некорректной работе функции. Далее в этой главе мы познакомимся с приемом, позволяющим проверить, правильное ли количество аргументов передано в функцию.

Заметьте, что в функции `print()` нет инструкции `return`, поэтому она всегда возвращает значение `undefined`, и не имеет смысла использовать ее в качестве части более сложного выражения. А функции `distance()` и `factorial()` могут вызываться в более сложных выражениях, что было показано в предыдущих примерах.

7.1.1. Вложенные функции

Стандарт ECMAScript v1 и реализации до JavaScript 1.2 допускали определение функций только в глобальном коде верхнего уровня. Однако JavaScript 1.2 и ECMAScript v3 допускают вложение определений функций в другие функции. Например:

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

Обратите внимание, что ECMAScript v3 не допускает произвольного расположения определений функций; они все еще ограничены глобальным кодом верхнего уровня и кодом функций верхнего уровня. Это значит, что определения функций не могут находиться, например, внутри циклов или условных инструкций.[1] Эти ограничения распространяются только на объявления функций с помощью инструкции `function`. Как мы обсудим далее в этой главе, функциональные литералы (другая возможность, введенная в JavaScript и стандартизованная в ECMAScript v3) могут присутствовать внут-

ри любых выражений JavaScript, а следовательно, и внутри `if` и других инструкций.

7.1.2. Конструктор Function()

Инструкция `function` – это не единственный способ определения новой функции. ECMAScript v1 и JavaScript 1.1 позволяют динамически определять функции с помощью конструктора `Function()` и оператора `new`. (Мы познакомились с оператором `new` в главе 5, а конструкторы подробно рассматриваются в главе 8.) Вот пример создания функции подобным образом:

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

Эта строка кода создает новую функцию, более или менее эквивалентную функции, определенной с помощью более привычного синтаксиса

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

Конструктор `Function()` принимает любое количество строковых аргументов. Последний аргумент – это тело функции. Он может содержать произвольные инструкции JavaScript, отделенные друг от друга точками с запятой. Все остальные аргументы конструктора представляют собой строки, задающие имена параметров определяемой функции. Если вы определяете функцию без аргументов, конструктору передается только одна строка – тело функции.

Заметьте, что конструктору `Function()` не передается какой-либо аргумент, задающий имя создаваемой им функции. Неименованные функции, созданные с помощью конструктора `Function()`, иногда называются *анонимными функциями*.

Вы можете задать резонный вопрос – зачем нужен конструктор `Function()`? Почему бы не определять все функции с помощью инструкции `function`? Одна причина состоит в том, что конструктор `Function()` позволяет динамически создавать и компилировать функции; он не ограничивает нас заранее скомпилированным телом функции в инструкции `function`. У этого преимущества есть и обратная сторона: конструктор `Function()` должен компилировать функцию при каждом ее вызове. Следовательно, вряд ли разумно помещать этот конструктор в тело цикла или внутрь часто вызываемой функции.

Другая причина для применения конструктора `Function()` заключается в том, что иногда возможность определить функцию как часть выражения JavaScript, а не как инструкцию, удобна, и такое решение даже изящно. Мы увидим примеры такого использования позднее в этой главе. В JavaScript 1.2, когда надо определить функцию в выражении, а не в инструкции, функцио-

---

[1] Различные реализации JavaScript могут иметь более ослабленные требования к определениям функций, чем требуется по стандарту. Например, реализации Netscape JavaScript 1.2 допускают «условные определения функций», указанные внутри инструкций `if`.

нальный литерал представляет собой еще более изящный вариант, чем конструктор Function(). Функциональные литералы мы рассмотрим в следующем разделе.

7.1.3. Функциональные литералы

ECMAScript v3 определяет, а JavaScript 1.2 реализует функциональные литералы, которые являются третьим способом создания функций. В главе 3 говорилось, функциональный литерал – это выражение, определяющее неименованную функцию. Синтаксис функционального литерала во многом похож на инструкцию function, за исключением того, что он используется как выражение, а не как инструкция, и ему не требуется имя функции. Следующие три строки кода определяют три более или менее идентичные функции с помощью инструкции function, конструктора Function() и функционального литерала:

```
function f(x) { return x*x; }              // предложение function
var f = new Function("x", "return x*x;");  // конструктор Function()
var f = function(x) { return x*x; };       // функциональный литерал
```

Функциональные литералы создают неименованные функции, но синтаксис допускает указание имени функции, что может быть полезно при написании рекурсивных функций, вызывающих самих себя. Например:

```
var f = function fact(x) { if (x <= 1) return 1; else return x*fact(x-1); };
```

Эта строка кода определяет неименованную функцию и сохраняет ссылку на нее в переменной f. Она на самом деле не создает функцию с именем fact(), но позволяет телу функции ссылаться с помощью этого имени на саму себя. Заметим, однако, что этот тип именованного функционального литерала не реализован правильно до JavaScript 1.5.

Помните, что инструкция function доступна во всех версиях JavaScript, конструктор Function() – только в JavaScript 1.1 и более поздних версиях, а функциональные литералы – только в JavaScript 1.2 и более поздних версиях. Вспомните, мы сказали, что три функции, определенные выше, «более или менее» эквивалентны, но есть и некоторые отличия между этими тремя способами, и мы рассмотрим эти отличия в разделе 11.5 «Конструктор Function() и функциональные литералы».

Функциональные литералы полезны примерно в тех же ситуациях, что и функции, созданные конструктором Function(). Поскольку они создаются выражениями JavaScript, а не инструкциями, то могут использоваться более гибко, и это особенно подходит для функций, которые вызываются только один раз и не должны иметь имени. Например, функция, определенная с помощью выражения функционального литерала, может быть сохранена в переменной, передана другой функции или даже вызвана непосредственно:

```
a[0] = function(x) { return x*x; };  // Определяем функцию и сохраняем ее
                                     // в переменной
```

```
a.sort(function(a,b){return a-b;});  // Определяем функцию; передаем ее другой
                                     // функции
var tensquared = (function(x) {return x*x;})(10);  // Определяем и вызываем
```

Как и конструктор `Function()`, функциональные литералы создают неименованные функции и не сохраняют автоматически эти функции в свойствах. Однако функциональные литералы имеют важное преимущество перед конструктором `Function()`. Тело функции, созданной с помощью `Function()`, должно быть задано в виде строки, и не всегда удобно выражать таким образом тела длинных, сложных функций. С другой стороны, тело функционального литерала использует стандартный синтаксис JavaScript. Кроме того, функциональный литерал анализируется и компилируется только один раз, а код JavaScript, переданный в виде строки конструктору `Function()`, должен анализироваться и компилироваться при каждом вызове конструктора.

7.2. Функции как данные

Самые важные особенности функций заключаются в том, что они могут определяться и вызываться, что было показано в предыдущем разделе. Определение и вызов функций – это синтаксические средства JavaScript и большинства других языков программирования. Однако в JavaScript функции – это не только синтаксические конструкции, но и данные, а это означает, что они могут присваиваться переменным, храниться в свойствах объектов или элементах массивов, передаваться как аргументы функциями и т. д.[1]

Чтобы понять, как функции могут быть одновременно синтаксическими конструкциями и данными JavaScript, рассмотрим следующее определение функции:

```
function square(x) { return x*x; }
```

Это определение создает новый объект функции и присваивает его переменной `square`. Имя функции действительно нематериально – это просто имя переменной, содержащей функцию. Функция может быть присвоена другой переменной, и при этом работать так же, как и раньше:

```
var a = square(4);  // a содержит число 16
var b = square;     // Теперь b ссылается на ту же функцию, что и square
var c = b(5);       // c содержит число 25
```

Функции могут быть также присвоены не только глобальным переменным, но и свойствам объектов. В этом случае их называют методами:

```
var o = new Object;
```

[1] Это может не показаться столь интересным, если вы не знакомы с такими языками, как Java, в которых функции являются частью программы, но не могут программой управляться.

```
o.square = new Function("x", "return x*x");  // Обратите внимание:
                                             // конструктор Function()
y = o.square(16);                            // "y" равно 256
```

У функций даже не обязательно должны быть имена, например в случае присваивании их элементам массива:

```
var a = new Array(3);
a[0] = function(x) { return x*x; }  // Обратите внимание: функциональный литерал
a[1] = 20;
a[2] = a[0](a[1]);                  // a[2] содержит 400
```

Синтаксис вызова функции в последнем примере выглядит необычно, однако это вполне допустимый вариант применения оператора () в JavaScript!

В примере 7.2 подробно показано, что можно делать, когда функции выступают в качестве данных. Он демонстрирует, каким образом функции могут передаваться другим функциям, а также храниться в ассоциативных массивах (представленных в главе 3 и подробно описанных в главе 8). Этот пример может быть несколько сложен, но комментарии объясняют, что происходит, и он стоит тщательного изучения.

Пример 7.2. Использование функций как данных

```
// Здесь мы определяем несколько простых функций
function add(x,y) { return x + y; }
function subtract(x,y) { return x - y; }
function multiply(x,y) { return x * y; }
function divide(x,y) { return x / y; }

// Эта функция, принимающая одну из вышеприведенных функций
// в качестве аргумента, и вызывающая ее для двух операндов
function operate(operator, operand1, operand2)
{
    return operator(operand1, operand2);
}

// Вот так можно вызвать эту функцию для вычисления значения выражения
// (2+3) + (4*5):
Var i = operate(add, operate(add, 2, 3), operate(multiply, 4, 5));

// Ради примера, мы реализуем эти функции снова, в этот раз с помощью
// функциональных литералов. Мы сохраним функции в ассоциативном массиве.
var operators = new Object();
operators["add"] = function(x,y) { return x+y; };
operators["subtract"] = function(x,y) { return x-y; };
operators["multiply"] = function(x,y) { return x*y; };
operators["divide"] = function(x,y) { return x/y; };
operators["pow"] = Math.pow;  // Для предопределенных функций это тоже работает

// Эта функция принимает имя оператора, ищет оператор в массиве, а затем
// вызывает его для переданных операндов. Обратите внимание на синтаксис
// вызова функции-оператора.
```

```
Function operate2(op_name, operand1, operand2)
{
    if (operators[op_name] == null) return "неизвестный оператор";
    else return operators[op_name](operand1, operand2);
}

// Вот так мы можем вызвать эту функцию для вычисления значения
// ("hello" + " " + "world"):
var j = operate2("add", "hello", operate2("add", " ", "world"))
// Используем предопределенную функцию Math.pow():
var k = operate2("pow", 10, 2)
```

Если предыдущий пример не убедил вас, что возможность передачи функций в качестве аргументов другим функциям и других способов использования функций как значений удобна, обратите внимание на функцию `Array.sort()`. Она сортирует элементы массива. Существует много возможных порядков сортировки (числовой, алфавитный, по датам, по возрастанию, по убыванию и т. д.), поэтому функция `sort()` принимает в качестве необязательного аргумента другую функцию, которая сообщает о том, как выполнять сортировку. Эта функция делает простую работу – получает два элемента массива, сравнивает их, а затем возвращает результат, указывающий, какой из элементов должен быть первым. Этот аргумент функции делает метод `Array.sort()` совершенно универсальным и бесконечно гибким – он может сортировать любой тип данных в любом мыслимом порядке!

7.3. Область видимости функции: объект вызова

Как говорилось в главе 4, тело функции JavaScript исполняется в локальной области видимости, отличной от глобальной. Эта новая область видимости создается путем добавления объекта вызова в начало цепочки областей видимости. Поскольку объект вызова является частью цепочки, любые свойства этого объекта доступны в теле функции как переменные. Локальные переменные, объявленные с помощью инструкции `var`, создаются как свойства этого объекта; параметры функции также становятся доступны как его свойства.

Кроме локальных переменных и параметров объект вызова определяет одно специальное свойство под названием `arguments`. Это свойство ссылается на другой специальный объект, известный как объект Arguments, обсуждаемый в следующем разделе. Так как свойство `arguments` – это свойство объекта вызова, оно имеет точно такой же статус, как локальные переменные и параметры функции. По этой причине идентификатор `arguments` должен рассматриваться как зарезервированное слово и не должен употребляться в качестве имени переменной или параметра.

7.4. Аргументы функции: объект Arguments

В теле функции идентификатор `arguments` всегда имеет особый смысл. `arguments` – это специальное свойство объекта вызова, ссылающееся на объект, известный как *объект Arguments*. Объект Arguments – это нечто вроде массива, позволяющего извлекать переданные функции значения по номеру, но не являющееся объектом Array. Объект Arguments также определяет дополнительное свойство `callee`, которое будет описано позднее.

Хотя функция JavaScript определяется с фиксированным количеством именованных аргументов, при вызове ей может быть передано любое их число. Массив `arguments[]` обеспечивает доступ к значениям аргументов, даже если у некоторых из них нет имени. Предположим, вы определили функцию `f`, которая требует один аргумент, `x`. Если вы вызовете эту функцию с двумя аргументами, то первый будет доступен в функции по имени параметра `x` или как `arguments[0]`. Второй аргумент доступен только как `arguments[1]`. Кроме того, как и у всех массивов, у `arguments` имеется свойство `length`, указывающее на количество содержащихся в массиве элементов. То есть в теле нашей функции `f`, вызываемой с двумя аргументами, `arguments.length` имеет значение 2.

Массив `arguments[]` полезен в нескольких отношениях. Следующий пример показывает, как с его помощью проверить, была ли функция вызвана с правильным числом аргументов, – ведь JavaScript этого за вас не сделает:

```
function f(x, y, z)
{
    // Сначала проверяем, правильное ли количество аргументов было передано
    if (arguments.length != 3) {
        throw new Error("функция f вызвана с " + arguments.length +
                        "аргументами, а требуется 3.");
    }
    // А теперь сам код функции...
}
```

Массив `arguments[]` также открывает важную возможность для функций JavaScript: они могут быть написаны таким образом, что будут работать с любым количеством аргументов. Вот пример, показывающий, как можно написать простую функцию `max()`, принимающую любое число аргументов и возвращающую значение самого большого из них (см. также встроенную функцию `Math.max()`, которая в ECMAScript v3 тоже принимает любое число аргументов):

```
function max()
{
    var m = Number.NEGATIVE_INFINITY;

    // Цикл по всем аргументам, поиск и сохранение
    // наибольшего из них
    for(var i = 0; i < arguments.length; i++)
```

```
        if (arguments[i] > m) m = arguments[i];
    // Возвращаем максимальный
    return m;
}

var largest = max(1, 10, 100, 2, 3, 1000, 4, 5, 10000, 6);
```

Массив arguments[] можно также использовать для написания функций, ожидающих фиксированное число именованных аргументов, за которыми следует произвольное число неименованных.

До сих пор в этом разделе мы говорили о «массиве arguments». Однако следует помнить, что arguments фактически не является массивом – это объект Arguments. В каждом объекте Arguments имеются пронумерованные элементы массива и свойство length, но с технической точки зрения это не массив. Лучше рассматривать его как объект, имеющий некоторые пронумерованные свойства. Спецификация ECMAScript не требует от объекта Arguments реализации какого-либо специфического для массива поведения. Хотя, например, можно присвоить значение свойству arguments.length, ECMAScript не требует этого для реального изменения числа элементов массива, определенных в объекте. (В главе 9 приведено описание особого поведения свойства length для настоящих объектов Array.)

У объекта Arguments есть одна *очень* необычная особенность. Когда у функции имеются именованные аргументы, элементы массива объекта Arguments являются синонимами локальных переменных, содержащих аргументы функции. Массив arguments[] и именованные аргументы – это два разных способа обращения к одной переменной. Изменение значения аргумента с помощью имени аргумента изменяет значение, извлекаемое через массив arguments[]. Изменение значения аргумента через массив arguments[] изменяет значение, извлекаемое по имени аргумента. Например:

```
function f(x) {
    alert(x);             // Показывает начальное значение аргумента
    arguments[0] = null;  // Изменяя элементы массива, мы также изменяем x
    alert(x);             // Теперь показывает "null"
}
```

7.4.1. Свойство callee

Кроме элементов своего массива, объект Arguments определяет свойство callee, ссылающееся на исполняемую в данный момент функцию. Это полезно, например, для рекурсивного вызова неименованных функций. Вот пример неименованного функционального литерала, вычисляющего факториал:

```
function(x) {
    if (x <= 1) return 1;
    return x * arguments.callee(x-1);
}
```

Свойство callee определено в ECMAScript v1 и реализовано в JavaScript 1.2.

7.5. Свойства и методы функции

Мы видели, что функции могут использоваться как значения в программах JavaScript и могут создаваться с помощью конструктора `Function()`. Это явные признаки того, что фактически функции представляют собой объект JavaScript, объект Function. А раз функции являются объектами, то они имеют свойства и методы – так же как, например, объекты String и Date. Теперь, после того как мы обсудили в контексте вызова функции объект вызова и объект Arguments, обратимся к самому объекту Function.

7.5.1. Свойство length

Как мы видели, в теле функции свойство `length` массива `arguments` задает количество аргументов, переданных этой функции. Однако свойство `length` самой функции имеет другой смысл. Это доступное только для чтения свойство возвращает количество аргументов, которое функция ожидает – то есть количество параметров, объявленных в ее списке параметров. Вспомним, что функция может вызываться с любым количеством аргументов, которые могут быть извлечены через массив `arguments`, независимо от количества объявленных параметров. Свойство `length` объекта Function в точности определяет, сколько объявленных параметров имеется у функции. Обратите внимание, что в отличие от `arguments.length`, это свойство `length` доступно как внутри, так и вне тела функции.

Следующий код определяет функцию с именем `check()`, получающую массив аргументов от другой функции. Она сравнивает свойство `arguments.length` со свойством `Function.length` (доступным как `arguments.callee.length`) и проверяет, передано ли функции столько аргументов, сколько она ожидает. Если это не так, генерируется исключение. За функцией `check()` следует тестовая функция `f`, демонстрирующая использование функции `check()`:

```
function check(args) {
    var actual = args.length;           // Фактическое число аргументов
    var expected = args.callee.length;  // Ожидаемое число аргументов
    if (actual != expected) {  // Если числа не совпадают, генерируется
                               // исключение
        throw new Error("неверное число аргументов: ожидается: " +
                        expected + "; фактически передано " + actual);
    }
}

function f(x, y, z) {
    // Проверим, соответствует ли ожидаемому фактическое количество аргументов
    // Если не соответствует, генерируем исключение
    check(arguments);
    // Теперь выполняем оставшуюся часть функции обычным образом
    return x + y + z;
}
```

Свойство `length` объекта Function стандартизовано ECMAScript v1 и реализовано в JavaScript 1.1 и более поздних версиях.[1]

7.5.2. Свойство prototype

Любая функция имеет свойство `prototype`, ссылающееся на предопределенный *объект-прототип*. Этот объект-прототип вступает в игру, когда функция используется в качестве конструктора с оператором `new`; он играет важную роль в процессе определения новых типов объектов. Мы подробно изучим это свойство в главе 8.

7.5.3. Определение собственных свойств функций

Когда функции требуется переменная, значение которой должно сохраняться между ее вызовами, часто оказывается удобным свойство объекта Function, позволяющее не забивать пространство имен определением глобальной переменной. Предположим, что надо написать функцию, возвращающую уникальный идентификатор при каждом своем вызове. Функция никогда не должна возвращать одно и то же значение дважды. Чтобы обеспечить это, функция должна запоминать уже возвращенные значения, и эта информация должна сохраняться между ее вызовами. Мы можем сохранить это значение в глобальной переменной, но это не нужно, так как информация используется только самой функцией. Лучше сохранить значение в свойстве объекта Function. Вот пример, который возвращает уникальное целое значение при каждом вызове:

```
// Создаем и инициализируем "статическую" переменную.
// Объявления функций обрабатываются до исполнения кода, поэтому
// мы действительно можем выполнить это присваивание до объявления
// функции
uniqueInteger.counter = 0;

// Вот сама функция. Она возвращает разные значения при каждом
// вызове и использует собственное "статическое" свойство для отслеживания
// последнего возвращенного значения.
function uniqueInteger() {
    // Инкрементируем и возвращаем "статическую" переменную
    return uniqueInteger.counter++;
}
```

7.5.4. Методы apply() и call()

В ECMAScript v3 есть два метода, определенные для всех функций, — `call()` и `apply()`. Эти методы позволяют вызывать функцию так, будто она является методом некоторого другого объекта. (Обратите внимание, что мы пока не

[1] В Netscape 4.0 ошибка приводит к некорректной работе этого свойства, если только атрибут `language` тега `<script>` явно не установлен в «JavaScript 1.2».

обсуждали методы; этот раздел может показаться более понятным после прочтения главы 8). Первый аргумент методов `call()` и `apply()` – это объект, для которого выполняется функция; этот аргумент становится значением ключевого слова `this` в теле функции. Все оставшиеся аргументы `call()` – это значения, передаваемые вызываемой функции. Так, чтобы передать функции `f()` два числа и вызвать ее как метод объекта `o`, можно использовать следующий код:

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

Это аналогично следующим строкам кода:

```
o.m = f;
o.m(1,2);
delete o.m;
```

Метод `apply()` похож на метод `call()`, за исключением того, что передаваемые функции аргументы задаются в виде массива:

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

Например, чтобы найти наибольшее число в массиве чисел, можно вызвать метод `apply()` для передачи элементов массива функции `Math.max()`:[1]

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

Метод `apply()` реализован в JavaScript 1.2, а метод `call()` не реализован до JavaScript 1.5.

[1] В этом примере мы предполагаем, что используется функция `Math.max()` из ECMAScript v3, принимающая произвольное число аргументов; версия ECMAScript v1 этой функции принимает только два аргумента.

8

Объекты

В главе 3 объясняется, что объекты – это один из фундаментальных и наиболее важных типов данных в JavaScript. В этой главе мы детально рассмотрим объекты JavaScript. Основы использования объектов, описанные в следующем разделе, достаточно просты, но как мы увидим в дальнейших разделах, применение и поведение объектов может быть более сложным.

8.1. Объекты и свойства

Объекты – это составные типы данных: они объединяют несколько значений в единый модуль и позволяют нам записывать и сохранять эти значения по имени. Другими словами, объект – это неупорядоченная коллекция *свойств*, каждое из которых имеет имя и значение. Именованные значения, содержащиеся в объекте, могут быть как элементарными типами, например числами или строками, так и другими объектами.

8.1.1. Создание объектов

Объекты создаются с помощью оператора `new`. За этим оператором должно быть указано имя функции-конструктора, служащей для инициализации объекта. Например, пустой объект (объект без свойств) создается так:

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

JavaScript поддерживает другие встроенные функции-конструкторы, инициализирующие созданные объекты менее тривиальными способами. Например, конструктор `Date()` инициализирует объект, представляющий дату и время:

```
var now = new Date();                          // Текущие дата и время
var new_years_eve = new Date(2000, 11, 31);  // Обозначает 31 декабря 2000
```

Позднее в этой главе мы увидим, что можно определять собственные методы-конструкторы, инициализирующие новые объекты нужным вам способом.

Объектные литералы предоставляют другой способ создания и инициализации новых объектов. Как мы видели в главе 3, объектные литералы позволяют встраивать описания объекта в код JavaScript в виде литерала, во многом так же, как мы встраиваем в код JavaScript текстовые данные в виде строк в кавычках. Объектный литерал представляет собой заключенный в фигурные скобки список спецификаций свойств, разделенных запятыми. Каждая спецификация свойства в объектном литерале состоит из имени свойства, за которым следует двоеточие и значение свойства. Например:

```
var circle = { x:0, y:0, radius:2 }
var homer = {
              name: "Homer Simpson",
              age: 34,
              married: true,
              occupation: "plant operator",
              email: "homer@simpsons.com"
};
```

Синтаксис объектных литералов определен в спецификации ECMAScript v3 и реализован в JavaScript 1.2 и более поздних версиях.

8.1.2. Установка и получение свойств

Обычно для доступа к значениям свойств объекта применяется оператор `.` (точка). Значение в левой части оператора должно быть ссылкой на объект (обычно просто именем переменной, содержащим ссылку на объект). Значение в правой части оператора должно быть именем свойства. Это должен быть идентификатор, а не строка или выражение. Так, обратиться к свойству `p` объекта `o` можно посредством выражения `o.p`, а к свойству `radius` объекта `circle` — `circle.radius`. Свойства объекта работают как переменные: в них (в свойствах) можно сохранять значения и считывать их. Например:

```
// Создаем объект. Сохраняем ссылку на него в переменной.
var book = new Object();

// Устанавливаем свойство в объекте.
book.title = "JavaScript: Наиболее полное руководство"

// Устанавливаем другие свойства. Обратите внимание на вложенные объекты.
book.chapter1 = new Object();
book.chapter1.title = "Введение в JavaScript";
book.chapter1.pages = 19;
book.chapter2 = { title: "Лексическая структура", pages: 6 };
// Читаем значения некоторых свойств из объекта.
```

```
alert("Заголовок: " + book.title + "\n\t" +
      "Глава 1 " + book.chapter1.title + "\n\t" +
      "Глава 2 " + book.chapter2.title);
```

Важно обратить внимание на один момент в этом примере – новое свойство объекта можно создать, просто присвоив ему значение. Если переменные должны быть объявлены с помощью ключевого слова var, то для свойств объекта нет необходимости (и способа) делать это. К тому же, после того как вы создадите свойство объекта, присвоив ему значение, в любой момент можно изменить значение свойства, просто присвоив ему новое значение:

```
book.title = "JavaScript: Книга с носорогом"
```

8.1.3. Перечисление свойств

Цикл for/in, который обсуждался в главе 6, предоставляет способ осуществления цикла по свойствам объекта, или их перечисления. Это может быть полезно при отладке сценариев или при работе с объектами, которые могут иметь произвольные свойства, имена которых заранее не известны. Следующий код показывает функцию, позволяющую вывести список имен свойств объекта:

```
function DisplayPropertyNames(obj) {
    var names = "";
    for(var name in obj) names += name + "\n";
    alert(names);
}
```

Обратите внимание, что цикл for/in не перечисляет свойства в каком-либо определенном порядке, и, несмотря на то что он перечисляет все свойства, определенные пользователем, он не перечисляет некоторые предопределенные свойства и методы.

8.1.4. Неопределенные свойства

Если попытаться прочитать значение несуществующего свойства (свойства, которому никогда не присваивалось значение), будет получено значение undefined (неопределенное значение, см. главу 3).

Для удаления свойства объекта предназначен оператор delete:

```
delete book.chapter2;
```

Обратите внимание, что при удалении свойства его значение не просто устанавливается в undefined; оператор delete действительно удаляет свойство из объекта. Цикл for/in демонстрирует это отличие: он перечисляет свойства, которым было присвоено значение undefined, но не перечисляет удаленные свойства.

8.2. Конструкторы

Ранее мы видели, что создавать и инициализировать новый объект JavaScript можно при помощи оператора `new` в сочетании с предопределенными функциями-конструкторами, такими как `Object()`, `Date()` и `Function()`. Эти предопределенные конструкторы и встроенные типы объектов, создаваемые ими, во многих случаях полезны. Однако в объектно-ориентированном программировании принято также работать с собственными объектными типами, определенными программой. Например, при написании программы, работающей с прямоугольниками, удобно было бы представить их в виде специального типа, или *класса* объектов. Каждый объект этого класса Rectangle должен иметь свойство `width` и `height`, так как это основные характеристики прямоугольников.

Чтобы создавать объекты со свойствами вроде только что упомянутых `width` и `height`, нам потребуется написать конструктор, создающий новый объект и инициализирующий его свойства. Конструктор – это функция JavaScript, имеющая две особенности:

- Она вызывается с помощью оператора `new`.
- Ей передается в виде значения ключевого слова `this` ссылка на только что созданный, пустой объект, и конструктор отвечает за выполнение соответствующих действий для нового объекта.

Пример 8.1 показывает, как можно определить и вызвать функцию-конструктор для объекта Rectangle.

Пример 8.1. Функция-конструктор объекта Rectangle

```
// Определяем конструктор.
// Обратите внимание, как в нем инициализируется объект с помощью
//  "this"
function Rectangle(w, h)
{
    this.width = w;
    this.height = h;
}

// Вызываем конструктор для создания двух объектов Rectangle.
// Мы передаем ширину и высоту конструктору, чтобы
// можно было правильно проинициализировать оба новых объекта.
var rect1 = new Rectangle(2, 4);
var rect2 = new Rectangle(8.5, 11);
```

Обратите внимание, как конструктор использует свои аргументы для инициализации объекта, на который ссылается ключевое слово `this`. Помните, что функция-конструктор просто инициализирует указанный объект; она не должна возвращать его.

Мы определили класс объектов, просто определив соответствующую функцию-конструктор – все объекты, созданные конструктором `Rectangle()`, те-

перь гарантированно имеют инициализированные свойства width и height. Это значит, что мы можем писать программы, учитывающие это обстоятельство и единообразно работающие со всеми объектами Rectangle. Каждый конструктор определяет класс объектов, поэтому стилистически важно дать функции-конструктору имя, указывающее на класс создаваемых ею объектов. Создание прямоугольника с помощью new Rectangle(1,2) выглядит намного понятнее, чем, например, new init_rect(1,2).

Функции-конструкторы обычно не возвращают значений. Они инициализируют объект, переданный как значение this, и ничего не возвращают. Однако конструктор имеет право возвращать объектное значение, и, если это так, этот возвращаемый объект становится значением выражения new. В этом случае объект, который был значением this, просто отбрасывается.

8.3. Методы

Метод – это функция JavaScript, вызываемая через объект. Вспомните, что функции – это значения и что в имени, с которым определена функция, нет ничего особенного – функция может быть присвоена любой переменной или даже любому свойству объекта. Если у нас есть функция f и объект o, мы можем определить метод с именем m, написав следующее:

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

Определив метод m() объекта o, мы сможем вызывать его так:

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

А вот как можно вызвать m(), если он требует двух аргументов:

```
o.m(x, x+2);
```

Методы имеют одну очень важную особенность: объект, через который вызывается метод, становится значением ключевого слова this в теле этого метода. Например, когда мы вызываем o.m(), тело метода ссылается на объект o с помощью ключевого слова this.

Прежде чем обсуждать ключевое слово this, выясним, зачем вообще нужны методы. Любая функция, используемая в качестве метода, обязательно получает дополнительный аргумент – объект, через который она вызвана. Обычно метод выполняет некоторую операцию над объектом, поэтому синтаксис вызова метода – очень изящный способ выразить тот факт, что функция работает с объектом. Сравните следующие две строки кода:

```
rect.setSize(width, height);
setRectSize(rect, width, height);
```

Эти две строки могут выполнять абсолютно одну и ту же операцию с объектом rect, но синтаксис вызова метода в первой строке более четко показывает, что направлена операция именно на объект rect, или на ее *цель*. (Если

синтаксис первой строки не кажется вам более естественным, то вы, видимо, новичок в объектно-ориентированном программировании. Когда вы приобретете некоторый опыт, оно вам понравится !)

Хотя и полезно рассматривать функции и методы по-разному, между ними не так много отличий, как может показаться на первый взгляд. Вспомните, что функции – это значения, хранящиеся в переменных, а переменные – это свойства глобального объекта. Следовательно, вызывая функцию, вы фактически вызываете метод глобального объекта. Внутри такой функции ключевое слово `this` ссылается на глобальный объект. Реальное отличие состоит в цели и назначении: методы пишутся для выполнения каких-либо операций с объектом `this`, а функции обычно работают изолированно и не используют объект `this`.

Типичное применение методов более ясно иллюстрируется в примере 8.2, в котором мы возвращаемся к объектам Rectangle и показываем, как определить и вызвать метод, работающий с объектами Rectangle.

Пример 8.2. Определение и вызов метода

```
// Эта функция использует ключевое слово this, поэтому нет смысла
// вызывать ее саму по себе, она должна быть сделана методом какого-
// либо объекта, в котором определены свойства «width» и «height».
function compute_area()
{
    return this.width * this.height;
}

// Создаем новый объект Rectangle с помощью конструктора, определенного ниже
var page = new Rectangle(8.5, 11);

// Определяем метод путем присваивания функции свойству объекта.
page.area = compute_area;

// Вызываем новый метод:
var a = page.area();    // a = 8.5*11 = 93.5
```

Один недостаток в примере 8.2 очевиден: перед тем как вы сможете вызвать метод `area()` объекта `rect`, необходимо присвоить этот метод свойству объекта. Несмотря на то что мы можем вызвать метод `area()` для конкретного объекта с именем `page`, мы не можем вызвать его для других объектов Rectangle, не присвоив им заранее этот метод. Это вам быстро надоест. Пример 8.3 определяет некоторые другие методы и показывает, как с помощью функции-конструктора они могут быть автоматически присвоены всем объектам Rectangle.

Пример 8.3. Определение методов в конструкторе

```
// Сначала определим некоторые функции, которые будут использоваться как методы
function Rectangle_area() { return this.width * this.height; }
function Rectangle_perimeter() { return 2*this.width + 2*this.height; }
function Rectangle_set_size(w,h) { this.width = w; this.height = h; }
```

```
function Rectangle_enlarge() { this.width *= 2; this.height *= 2; }
function Rectangle_shrink() { this.width /= 2; this.height /= 2; }

// Затем определим метод-конструктор для наших объектов Rectangle
// Конструктор инициализирует свойства, а также присваивает методы.
function Rectangle(w, h)
{
    // Инициализация свойств объекта.
    this.width = w;
    this.height = h;

    // Определение методов объекта.
    this.area = Rectangle_area;
    this.perimeter = Rectangle_perimeter;
    this.set_size = Rectangle_set_size;
    this.enlarge = Rectangle_enlarge;
    this.shrink = Rectangle_shrink;
}
// Теперь мы можем вызывать методы сразу после создания объекта Rectangle.
var r = new Rectangle(2,2);
var a = r.area();
r.enlarge();
var p = r.perimeter();
```

Прием, показанный в примере 8.3, также имеет недостаток. В этом примере конструктор `Rectangle()` устанавливает семь свойств для каждого инициализируемого им объекта Rectangle, даже несмотря на то, что пять из этих свойств имеют постоянные значения, одинаковые для всех прямоугольников. Каждое из свойств занимает место в памяти; добавив методы к нашему классу Rectangle, мы более чем втрое увеличили требования к памяти, занимаемой каждым объектом Rectangle. К счастью, в JavaScript есть возможность обойти это препятствие: объекту позволено наследовать свойства от объекта-прототипа. В следующем разделе эта технология рассматривается подробно.

8.4. Прототипы и наследование

Мы видели, насколько неэффективным может быть применение конструктора для присваивания методов инициализируемому объекту. При этом каждый созданный конструктором объект имеет идентичные копии одних и тех же свойств-методов. Есть значительно более эффективный способ задания методов, констант и других свойств, общих для всех объектов класса.

Объекты JavaScript «наследуют» свойства от *объекта-прототипа*.[1] Каждый объект имеет прототип; все свойства объекта-прототипа становятся

[1] Прототипы введены в JavaScript 1.1, они не поддерживаются ныне устаревшим JavaScript 1.0.

свойствами любых объектов, для которых он является прототипом. То есть любой объект *наследует* свойства от своего прототипа.

Прототип объекта определяется функцией-конструктором, используемой для создания и инициализации объекта. Все функции в JavaScript имеют свойство `prototype`, ссылающееся на объект. Этот объект-прототип изначально пуст, однако любые свойства, которые вы определите в нем, будут наследоваться всеми объектами, созданными с помощью конструктора.

Конструктор определяет класс объектов и инициализирует свойства, такие как `width` и `height`, являющиеся для класса переменными состояния. Объект-прототип связан с конструктором, так что каждый член класса наследует тот набор свойств, который имеется в прототипе. Это значит, что объект-прототип – идеальное место для методов и других свойств-констант.

Обратите внимание, что наследование осуществляется автоматически, как часть процесса поиска значения свойства. Свойства *не* копируются из объекта-прототипа в новый объект; они просто присутствуют, как если бы они были свойствами этих объектов. Это имеет два важных последствия. Во-первых, использование объектов-прототипов может в значительной степени уменьшить объем памяти, требуемый для каждого объекта, так как объекты могут наследовать многие из своих свойств. Во-вторых, объект наследует свойства, даже если они были добавлены в его прототип *после* создания объекта.

У каждого класса имеется один объект-прототип, с одним наборов свойств. Но потенциально может существовать много экземпляров класса, каждый из которых наследует свойства прототипа. Свойство прототипа может наследоваться многими объектами, поэтому JavaScript должен обеспечить фундаментальную асимметричность между чтением и записью значений свойств. Когда вы читаете свойство `p` объекта `o`, JavaScript сначала проверяет, есть ли у объекта `o` свойство с именем `p`. Если такого свойства нет, то проверяется, есть ли свойство с именем `p` в объекте-прототипе. Так работает наследование на базе прототипов.

С другой стороны, когда свойству присваивается значение, JavaScript не использует объект-прототип. Чтобы понять, почему это так, подумайте, что произошло бы в этом случае: предположим, вы пытаетесь установить значение свойства `o.p`, а у объекта `o` нет свойства с именем `p`. Предположим теперь, что JavaScript идет дальше и ищет свойство `p` в объекте-прототипе объекта `o` и позволяет вам изменить значение свойства прототипа. Но вы изменили значение `p` для всего класса объектов, а это вовсе не то, что вам требовалось.

Поэтому наследование свойств происходит только при чтении значения свойств, но не при записи в них. Если вы устанавливаете свойство `p` в объекте `o`, который наследует это свойство от своего прототипа, происходит создание нового свойства непосредственно в объекте `p`. Теперь, когда объект `o` имеет собственное свойство с именем `p`, он больше не наследует значение `p` от прототипа. Когда вы читаете значение `p`, JavaScript сначала ищет его в свой-

ствах объекта o. Так как он находит свойство p, определенное в o, ему не требуется искать его в объекте-прототипе, и JavaScript никогда не найдет значение p, определенное в нем. Мы иногда говорим, что свойство p «затеняет» или «скрывает» свойство p объекта-прототипа. Наследование прототипов может показаться запутанным. Все вышеизложенное хорошо иллюстрирует рис. 8.1.

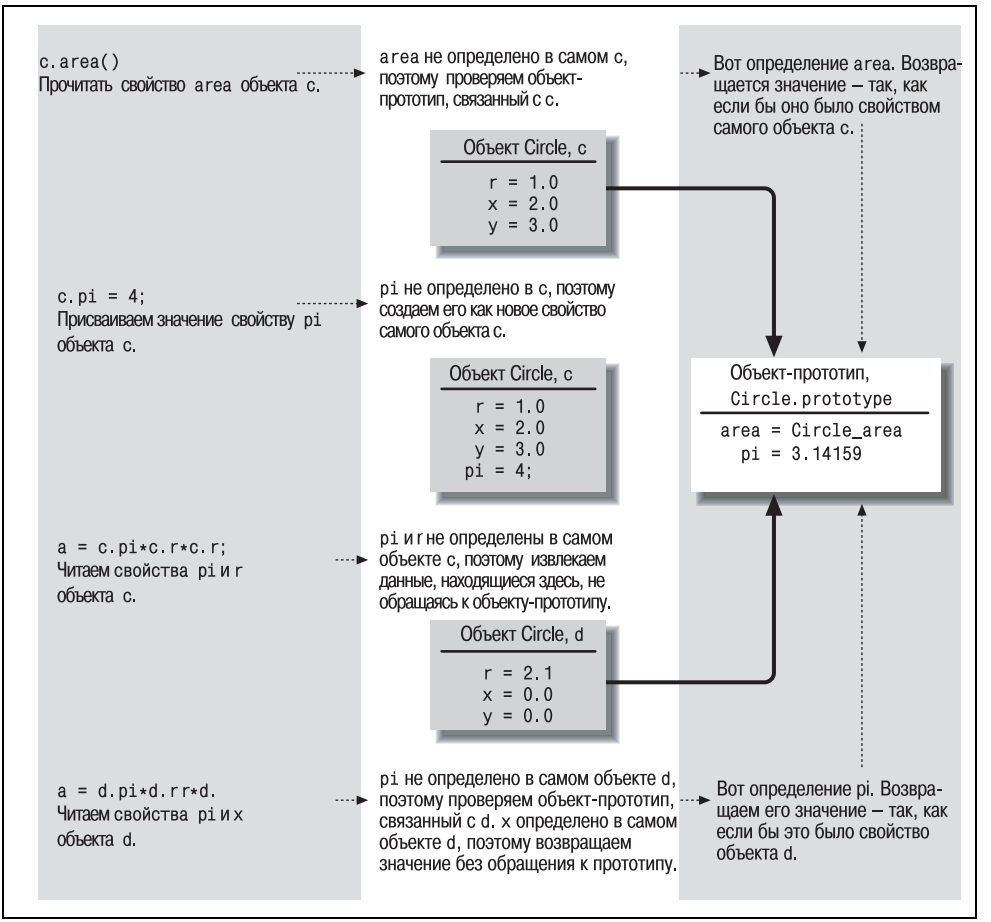

Рис. 8.1. *Объекты и прототипы*

Свойства прототипа совместно используются всеми объектами класса, поэтому, как правило, их имеет смысл применять только для определения свойств, совпадающих для всех объектов класса. Это делает прототипы идеальными для определения методов. Другие свойства с постоянными значениями (такими как математические константы) также подходят для определения в качестве свойств прототипа. Если класс определяет свойство с очень часто используемым значением по умолчанию, то можно определить это свойство и его значение по умолчанию в объекте-прототипе. Тогда те немно-

гие объекты, которые хотят изменить значение по умолчанию, могут создавать свои частные копии свойства и определять собственные значения, отличные от значения по умолчанию.

Перейдем от абстрактного обсуждения наследования прототипов к конкретному примеру. Предположим, что мы определили функцию-конструктор `Circle()` для создания объектов, представляющих круг. Объект-прототип для этого класса – `Circle.prototype`,[1] поэтому мы можем определить константу, доступную всем объектам Circle:

```
Circle.prototype.pi = 3.14159;
```

Полноценное воплощение этой идеи приведено в примере 8.4. Код определяет класс `Circle`, сначала определяя конструктор `Circle()` для инициализации каждого отдельного объекта, а затем устанавливая свойства `Circle.prototype` для определения методов и констант, совместно используемых всеми экземплярами класса.

Пример 8.4. Определение класса Circle с помощью объекта-прототипа

```
// Определяем метод-конструктор для нашего класса.
// Используем его для инициализации свойств, различных для каждого
// отдельного объекта Circle.
function Circle(x, y, r)
{
    this.x = x;  // Координата X центра круга.
    this.y = y;  // Координата X центра круга.
    this.r = r;  // Радиус круга.
}

// Создаем и отбрасываем первоначальный объект Circle.
// Это приводит к созданию объекта-прототипа в JavaScript 1.1.
new Circle(0,0,0);

// Определяем константу: свойство, которое будет совместно использоваться
// всеми объектами circle. На самом деле нам следовало бы использовать
```

---

[1] Объект-прототип для конструктора создается JavaScript автоматически. В большинстве версий JavaScript любой функции автоматически предоставляется пустой объект-прототип на тот случай, если она будет использоваться в качестве конструктора. Однако в JavaScript 1.1 объект-прототип не создается до тех пор, пока функция в первый раз не будет вызвана в качестве конструктора. Это значит, что если вам требуется совместимость с JavaScript 1.1, вы должны перед тем, как использовать объект-прототип для присваивания методов и констант объектам класса, создать как минимум один объект этого класса. Поэтому если мы определили конструктор `Circle()`, но пока не создавали с его помощью объекты Circle, то для определения константного свойства `pi` нам придется написать следующий код:

```
// Сначала создаем и отбрасываем фиктивный объект; это приводит к созданию объекта-прототипа.
new Circle;
// Теперь мы можем устанавливать свойства в прототипе.
Circle.prototype.pi = 3.14159
```

```
// Math.PI, но для целей обучения мы поступаем именно так.
Circle.prototype.pi = 3.14159;

// Определяем метод для вычисления длины окружности.
// Сначала объявляем функцию, а затем присваиваем ее свойству прототипа.
// Обратите внимание на использование константы, определенной выше.
function Circle_circumference() { return 2 * this.pi * this.r; }
Circle.prototype.circumference = Circle_circumference;

// Определяем еще один метод. В этот раз мы одновременно воспользуемся
// функциональным литералом для определения функции и присвоим его
// свойству прототипа.
Circle.prototype.area = function() { return this.pi * this.r * this.r; }

// Класс Circle определен.
// Теперь мы можем создать экземпляр класса и вызывать его методы.
var c = new Circle(0.0, 0.0, 1.0);
var a = c.area();
var p = c.circumference();
```

8.4.1. Прототипы и встроенные классы

Не только классы, определенные пользователем, имеют объекты-прототипы. Встроенные классы, такие как String и Date, также имеют объекты-прототипы, и вы можете присваивать им значения.[1] Например, следующий код определяет новый метод, доступный всем объектам String:

```
// Возвращает true, если последним символом является c
String.prototype.endsWith = function(c) {
    return (c == this.charAt(this.length-1))
}
```

Определив новый метод `endsWith()` в объекте-прототипе String, мы сможем обратиться к нему следующим образом:

```
var message = "hello world";
message.endsWith('h')  // Возвращает false
message.endsWith('d')  // Возвращает true
```

8.5. Объектно-ориентированный JavaScript

Хотя JavaScript поддерживает тип данных, который мы называем объектом, в нем нет формального понятия класса. Это сильно отличает его от классических объектно-ориентированных языков программирования, таких как C++ и Java. Общая черта объектно-ориентированных языков – это их сильная типизация и поддержка наследования на базе классов. По этому критерию JavaScript легко исключить из числа истинно объектно-ориенти-

[1] В JavaScript 1.1 и более поздних версиях.

рованных языков. С другой стороны, мы видели, что JavaScript активно использует объекты и имеет особый тип наследования на базе прототипов. JavaScript – это истинно объектно-ориентированный язык. Он был вдохновлен несколькими другими (малопонятными) объектно-ориентированными языками, в которых реализовано наследование на базе прототипов вместо наследования на основе классов.

Несмотря на то что JavaScript – это объектно-ориентированный язык, не базирующийся на классах, он неплохо имитирует возможности языков на базе классов, таких как Java и C++. Я употребил термин «класс» в этой главе неформально. В данном разделе проводятся более формальные параллели между JavaScript и истинным наследованием на базе класса в таких языках, как Java и C++.[1]

Начнем с того, что определим некоторые базовые термины. Объект, как мы уже видели, – это структура данных, содержащая различные фрагменты именованных данных, которая также может содержать методы для работы с этими фрагментами данных. Объект группирует связанные значения и методы в единый удобный набор, который, как правило, облегчает процесс программирования, увеличивая модульность и степень повторного использования кода. Объекты в JavaScript могут иметь произвольное число свойств, и свойства могут добавляться в объект динамически. В строго типизированных языках, таких как Java и C++, это не так. В них любой объект имеет предопределенный набор свойств,[2] а каждое свойство – предопределенный тип. Имитируя объектно-ориентированные приемы программирования при помощи объектов JavaScript, мы, как правило, заранее определяем набор свойств для каждого объекта и тип данных, содержащихся в каждом свойстве.

В Java и C++ *класс* определяет структуру объекта. Класс точно задает, какие поля содержатся в объекте и какие типы данных эти поля содержат. Он также определяет методы, работающие с объектом. В JavaScript нет формального понятия класса, но, как мы видели, в этом языке приближение к возможностям классов реализуется с помощью конструкторов и объектов-прототипов.

И JavaScript и объектно-ориентированные языки, основывающиеся на классах, допускают наличие множества объектов одного класса. Мы часто говорим, что объект – это *экземпляр* класса. Таким образом, может быть много экземпляров любого класса.

В Java обычная практика программирования заключается в присвоении классам имен с первой прописной буквой, а объектам – имен в нижнем регистре. Это соглашение помогает отличать в коде классы и объекты и этому соглашению полезно следовать при написании программ на языке Ja-

---

[1] Этот раздел следует прочитать даже тем, кто не знаком с такими языками и таким стилем объектно-ориентированного программирования.

[2] Обычно в Java и C++ они называются «полями», но здесь мы будем называть их свойствами, поскольку такая терминология принята в JavaScript.

vaScript. Например, в предыдущих разделах мы определили классы Circle и Rectangle и создали экземпляры этих классов с именами `c` и `rect`.

Члены Java-класса могут принадлежать к одному из четырех основных типов: свойства экземпляра, методы экземпляра, свойства класса и методы класса. В следующих разделах мы рассмотрим различия между этими типами и поговорим о том, как JavaScript имитирует эти типы.

8.5.1. Свойства экземпляра

Каждый объект имеет свои собственные копии *свойств экземпляра*. Другими словами, если имеется 10 объектов данного класса, то имеется и 10 копий каждого свойства экземпляра. Например, в нашем классе Circle любой объект Circle имеет свойство `r`, задающее радиус круга. В данном случае `r` представляет собой свойство экземпляра. А поскольку каждый объект имеет собственную копию свойства экземпляра, доступ к этим свойствам можно получить через отдельные объекты. Если, например, `c` – это объект, представляющий собою экземпляр класса Circle, мы ссылаемся на его радиус следующим образом:

```
c.r
```

По умолчанию любое свойство объекта в JavaScript является свойством экземпляра. Однако чтобы по-настоящему имитировать объектно-ориентированное программирование, мы будем говорить, что свойства экземпляра в JavaScript – это те свойства, которые создаются и/или инициализируются в объекте функцией-конструктором.

8.5.2. Методы экземпляра

Метод экземпляра во многом похож на свойство экземпляра, за исключением того, что это метод, а не значение. (В Java функции и методы не являются данными, как это имеет место в JavaScript, поэтому в Java данное различие выражено более четко.) Методы экземпляра вызываются по отношению к определенному объекту, или экземпляру. Метод `area()` нашего класса Circle представляет собой метод экземпляра. Он вызывается для объекта Circle следующим образом:

```
a = c.area();
```

Методы экземпляра ссылаются на объект или экземпляр, с которым они работают, при помощи ключевого слова `this`. Метод экземпляра может быть вызван для любого экземпляра класса, но это не значит, что каждый объект содержит собственную копию метода, как это делается для свойств экземпляра. Вместо этого каждый метод экземпляра совместно используется всеми экземплярами класса. В JavaScript мы определяем метод экземпляра для класса путем присваивания в конструкторе функции свойству объекта-прототипа. Так все объекты, созданные данным конструктором, совместно ис-

пользуют наследованную ссылку на функцию и могут вызывать ее с помощью приведенного выше синтаксиса вызова методов.

8.5.3. Свойства класса

Свойство класса в Java – это свойство, связанное с самим классом, а не с каждым экземпляром этого класса. Независимо от того, сколько создано экземпляров класса, есть только одна копия каждого свойства класса. Так же, как свойства экземпляра доступны через экземпляр класса, доступ к свойствам класса можно получить через сам класс. `Number.MAX_VALUE` – это пример свойства класса в JavaScript: свойство `MAX_VALUE` доступно через класс Number. Так как имеется только одна копия каждого свойства класса, свойства класса по существу являются глобальными. Однако их достоинство состоит в том, что они связаны с классом и имеют логичную нишу, позицию в пространстве имен JavaScript, где они вряд ли будут перекрыты другими свойствами с тем же именем. Ясно, что свойства класса имитируются в JavaScript просто определением свойства самой функции-конструктора. Например, создать свойство класса `Circle.PI` для хранения математической константы *pi* можно так:

```
Circle.PI = 3.14;
```

`Circle` – это функция-конструктор, но поскольку функции JavaScript представляют собой объекты, мы можем создать свойство функции точно так же, как можем создавать свойства любого другого объекта.

8.5.4. Методы класса

И наконец, мы подошли к методам класса. Метод класса – это метод, связанный с классом, а не с экземпляром класса; они вызываются через сам класс, а не через конкретный экземпляр класса. Метод `Date.parse()` (описываемый в справочном разделе, посвященном базовому JavaScript) – это метод класса. Он всегда вызывается через объект конструктора Date, а не через конкретный экземпляр класса Date.

Методы класса не вызываются через определенный объект, поэтому они не могут осмысленно использовать ключевое слово `this`, – ведь `this` ссылается на объект, для которого вызывается метод экземпляра. Как и свойства класса, методы класса являются глобальными. Методы класса не работают с конкретным экземпляром, поэтому их, как правило, проще рассматривать как функции, вызываемые через класс. Как и в случае со свойствами класса, связь этих функций с классом дает им удобную нишу в пространстве имен JavaScript и предотвращает возникновение конфликтов в пространстве имен. Для того чтобы определить метод класса в JavaScript, требуется сделать соответствующую функцию свойством конструктора.

8.5.5. Пример: класс Circle

Пример 8.5 – это еще одна реализация нашего класса Circle, содержащая примеры членов класса, относящихся к каждому из четырех базовых типов.

Пример 8.5. Определение свойств и методов экземпляра и класса

```
function Circle(radius) {   // Конструктор определяет сам класс.
    // r - это свойство экземпляра, определенное и инициализированное в
    // конструкторе.
    this.r = radius;
}

// Circle.PI - это свойство класса, свойство функции-конструктора.
Circle.PI = 3.14159;

// Это функция, вычисляющая площадь круга.
function Circle_area() { return Circle.PI * this.r * this.r; }

// Здесь мы делаем функцию методом экземпляра, присваивая ее
// объекту-прототипу конструктора.
// Внимание: в JavaScript 1.2 мы можем использовать функциональный
// литерал без присваивания ему имени Circle_area.
Circle.prototype.area = Circle_area;

// Еще одна функция. Она принимает в качестве аргументов два объекта
// Circle и возвращает тот из них, который больше другого (т.е. имеет
// больший радиус).
function Circle_max(a,b) {
    if (a.r > b.r) return a;
    else return b;
}

// Так как эта функция сравнивает два объекта Circle, нет смысла делать ее
// методом экземпляра, работающим с одиночным объектом Circle. Но мы также
// не хотим делать ее изолированной функцией, поэтому мы сделаем ее методом
// класса, присвоив ее функции-конструктору:
Circle.max = Circle_max;

// Вот примеры кода, использующие каждое из этих полей:
var c = new Circle(1.0);      // Создаем экземпляр класса Circle
c.r = 2.2;                    // Устанавливаем свойство экземпляра r
var a = c.area();             // Вызываем метод экземпляра area()
var x = Math.exp(Circle.PI);  // Используем свойство класса PI в собственных
                              // расчетах
var d = new Circle(1.2);      // Создаем еще один экземпляр Circle
var bigger = Circle.max(c,d); // Используем метод класса max()
```

8.5.6. Пример: комплексные числа

В примере 8.6 представлен еще один способ определения класса объектов в JavaScript, несколько более формальный, чем предыдущий. Код и комментарии достойны тщательного изучения. Обратите внимание, что здесь используется синтаксис функциональных литералов JavaScript 1.2. Так как пример требует данной версии языка (или более поздней), в нем конструктор не вызывается перед присваиванием его объекту-прототипу для обеспечения совместимости с JavaScript 1.1.

Пример 8.6. Класс комплексных чисел

```
/*
 * Complex.js:
 * В этом файле определяется класс Complex для представления комплексных
 * чисел. Вспомним, что комплексное число - это сумма вещественного
 * и мнимого числа, и что мнимое число i - это квадратный
 * корень из -1.
 */

/*
 * Первый шаг в определении класса - это определение функции-конструктора
 * класса. Этот конструктор должен инициализировать все свойства
 * экземпляра объекта. Это неотъемлемые «переменные состояния»,
 * делающие каждый экземпляр класса различным.
 */
function Complex(real, imaginary) {
    this.x = real;       // Вещественная часть числа
    this.y = imaginary;  // Мнимая часть числа
}

/*
 * Второй шаг в определении класса - это определение методов экземпляра
 * (и возможно других свойств) в объекте-прототипе конструктора. Любые
 * свойства, определенные в этом объекте, будут унаследованы всеми
 * экземплярами класса. Обратите внимание, что методы экземпляра
 * неявно работают с ключевым словом this. Для многих методов никаких
 * других аргументов не требуется.
 */

// Возвращает модуль комплексного числа. Он определяется как
// расстояние на комплексной плоскости до числа от начала координат (0,0).
Complex.prototype.magnitude = function() {
    return Math.sqrt(this.x*this.x + this.y*this.y);
};

// Возвращает комплексное число, противоположное данному.
Complex.prototype.negative = function() {
    return new Complex(-this.x, -this.y);
};
```

```
//  Преобразует объект Complex в строку в понятном формате.
//  Вызывается, когда объект Complex используется как строка.
Complex.prototype.toString = function() {
    return "{" + this.x + "," + this.y + "}";
};

// Возвращает вещественную часть комплексного числа. Эта функция
// вызывается, когда объект Complex рассматривается как числовое значение.
Complex.prototype.valueOf = function() { return this.x; }

/*
 * Третий шаг в определении класса – это определение методов класса,
 * констант, и других необходимых свойств класса как свойств самой
 * функции-конструктора (а не как свойств объекта-прототипа конструктора).
 * Обратите внимание, что методы класса не используют ключевое слово this:
 * они работают только со своими аргументами.
 */

// Складывает два комплексных числа и возвращает результат.
Complex.add = function (a, b) {
    return new Complex(a.x + b.x, a.y + b.y);
};

// Вычитает одно комплексное число из другого.
Complex.subtract = function (a, b) {
    return new Complex(a.x - b.x, a.y - b.y);
};

// Умножает два комлексных числа и возвращает полученное произведение.
Complex.multiply = function(a, b) {
    return new Complex(a.x * b.x - a.y * b.y,
                       a.x * b.y + a.y * b.x);
};

// Несколько полезных предопределенных комплексных чисел.
// Они определяются как свойства класса, где они могут использоваться
// как "константы".
// (Тем не менее на самом деле они доступны не только для чтения.)
Complex.zero = new Complex(0,0);
Complex.one = new Complex(1,0);
Complex.i = new Complex(0,1);
```

8.5.7. Надклассы и подклассы

В Java, C++ и других объектно-ориентированных языках на базе классов имеется явная концепция *иерархии классов*. Каждый класс может иметь *надкласс*, от которого он наследует свойства и методы. Любой класс может быть расширен или иметь подкласс, наследующий его поведение. Как мы видели, JavaScript поддерживает наследование прототипов вместо наследования на базе классов. Тем не менее в JavaScript могут быть проведены ана-

логии с иерархией классов. В JavaScript класс Object – это наиболее общий класс, и все другие классы являются его специализированными версиями, или подклассами. Можно также сказать, что Object – это надкласс всех встроенных классов. Все классы наследуют несколько базовых методов класса Object (описанных далее в этой главе).

Мы узнали, что объекты наследуют свойства от объекта-прототипа их конструктора. Как они могут наследовать свойства еще и от класса Object? Вспомните, что объект-прототип сам представляет собой объект; он создается с помощью конструктора `Object()`. Это значит, что объект-прототип наследует свойства от `Object.prototype`! Поэтому объект класса Complex наследует свойства от объекта `Complex.prototype`, который в свою очередь наследует свойства от `Object.prototype`. Когда вы ищете свойство в объекте Complex, сначала выполняется поиск в самом объекте. Если свойство не найдено, поиск далее продолжается в объекте `Complex.prototype`. И наконец, если свойство не найдено и в этом объекте, выполняется поиск в объекте `Object.prototype`.

Обратите внимание, что поскольку в объекте-прототипе Complex поиск происходит раньше, чем в объекте-прототипе Object, свойства объекта `Complex.prototype` скрывают любые свойства с тем же именем из `Object.prototype`. Так, в классе, показанном в примере 8.6, мы определили в объекте `Complex.prototype` метод `toString()`. `Object.prototype` также определяет метод с этим именем, но объекты Complex никогда не увидят его, поскольку определение `toString()` в `Complex.prototype` будет найдено раньше.

Все классы, которые мы показали в этой главе, представляют собой непосредственные подклассы Object. Это типично для программирования на JavaScript; обычно в создании более сложной иерархии классов нет никакой необходимости. Однако когда это требуется, можно создать подкласс любого другого класса. Предположим, что мы хотим создать подкласс класса Complex, чтобы добавить некоторые другие методы. Для этого мы просто должны быть уверены, что объект-прототип нового класса сам является экземпляром Complex, и потому наследует все свойства `Complex.prototype`:

```
// Это конструктор подкласса.
function MoreComplex(real, imaginary) {
    this.x = real;
    this.y = imaginary;
}

// Мы делаем этот прототип объектом класса Complex. Это значит, что
// экземпляры нашего нового класса будут наследоваться от MoreComplex.prototype,
// который наследуется от Complex.prototype, который, в свою очередь,
// будет наследником Object.prototype.
MoreComplex.prototype = new Complex(0,0);

// Теперь добавим новый метод или другие новые возможности в этот подкласс.
MoreComplex.prototype.swap = function() {
    var tmp = this.x;
```

```
    this.x = this.y;
    this.y = tmp;
}
```

У этой технологии создания подклассов есть один небольшой недостаток. Так как мы явно присвоили `MoreComplex.prototype` созданный нами объект, мы перекрыли объект-прототип, предоставленный JavaScript, и отбросили предоставленное нам значение свойства `constructor`. Свойство `constructor`, которое описано далее в этой главе, предназначено для ссылки на функцию-конструктор, создавшую объект. Однако объект `MoreComplex` не имеет собственного конструктора, а наследует свойство `constructor` своего надкласса. Одним из решений является явное присваивание значения этому свойству:

```
MoreComplex.prototype.constructor = MoreComplex;
```

Обратите внимание, что в JavaScript 1.1 свойство `constructor` доступно только для чтения и не может быть установлено подобным образом.

8.6. Объекты как ассоциативные массивы

Мы видели, как доступ к свойствам объекта осуществляется посредством оператора «точка». Доступ к этим свойствам возможен также при помощи оператора `[]`, который обычно применяется при работе с массивами. Таким образом, следующие два выражения JavaScript имеют одинаковое значение:

```
object.property
object["property"]
```

Важное различие между этими двумя синтаксисами, на которое следует обратить внимание, состоит в том, что в первом варианте имя свойства представляет собой идентификатор, а во втором – строку. Скоро мы узнаем, почему это так важно.

В Java, C, C++ и подобных языках с сильной типизацией объект может иметь только фиксированное число свойств, и имена этих свойств должны быть определены заранее. JavaScript – это язык со свободной типизацией, поэтому к нему данное правило не применяется, – программа может создавать любое количество свойств в любом объекте. Однако в случае применения оператор «точка» для доступа к свойству объекта имя свойства задается идентификатором. Идентификаторы должны быть частью текста программы на JavaScript – они не являются типом данных и ими нельзя манипулировать из программы.

С другой стороны, при обращении к свойству объекта с помощью нотации массивов `[]` имя свойства задается в виде строки. Строки – это тип данных JavaScript, поэтому они могут создаваться и изменяться во время работы программы. И поэтому можно, например, написать в JavaScript следующий код:

```
var addr = "";
```

```
for(i = 0; i < 4; i++) {
    addr += customer["address" + i] + '\n';
}
```

В этом коде читаются и конкатенируются свойства address0, address1, address2 и address3 объекта customer.

Этот короткий пример демонстрирует гибкость нотации массивов при обращении к свойствам объекта с помощью строковых выражений. Мы могли написать этот пример и с оператора «точка», но есть ситуации, где подойдет только нотация массивов. Предположим, что вы пишете программу, обращающуюся к сетевым ресурсам для вычисления текущей суммы инвестиций пользователя на фондовом рынке. Программа разрешает пользователю вводить названия любых имеющихся у него акций, а также количество каждого вида акций. Можно организовать хранение этой информации при помощи объекта с именем portfolio. Объект имеет по одному свойству для каждого вида акций. Имя свойства – это название акции, а значение свойства – количество акций данного вида. Другими словами, если, например, у пользователя имеется 50 акций IBM, то свойство portfolio.ibm имеет значение 50.

Частью этой программы должен быть цикл, запрашивающий у пользователя название имеющихся у него акций, а затем количество акций данного типа. Внутри цикла должен быть код, похожий на следующий:

```
var stock_name = get_stock_name_from_user();
var shares = get_number_of_shares();
portfolio[stock_name] = shares;
```

Пользователь вводит названия акций во время выполнения, поэтому нет способа узнать имена свойств заранее. А раз имена свойств при написании программы неизвестны, то доступа к свойствам объекта portfolio при помощи оператора «точка» невозможен. Однако можно обратиться к оператору [], так как в нем для имени свойства используется строковое значение (которое может изменяться во время выполнения), а не идентификатор (который должен быть задан непосредственно в тексте программы).

Когда объект используется в такой форме, его часто называют *ассоциативным массивом* – структурой данных, позволяющей связывать произвольные значения с произвольными строками. Объекты JavaScript на самом деле внутренне реализованы как ассоциативные массивы. Нотация «.» для доступа к свойствам делает их похожими на статические объекты C++ и Java, и они прекрасно работают в этой роли. Но они также предоставляют мощное средство для связи значений с произвольными строками. В этом отношении объекты JavaScript значительно больше похожи на массивы в Perl, чем на объекты C++ или Java.

В главе 6 был введен цикл for/in. Настоящая мощь этой конструкции JavaScript становится понятной, если рассмотреть ее применение с ассоциативными массивами. Возвращаясь к примеру с портфелем акций, после ввода пользователем данных по своему портфелю вычислить текущую общую стоимость последнего можно при помощи следующего кода:

```
var value = 0;
for (stock in portfolio) {
    // Для каждого вида акций в портфеле получаем стоимость одной акции
    // и умножаем ее на число акций.
    value += get_share_value(stock) * portfolio[stock];
}
```

Здесь не обойтись без цикла `for/in`, поскольку названия акций заранее не известны. Это единственный способ извлечения имен этих свойств из ассоциативного массива (или объекта JavaScript) по имени `portfolio`.

8.7. Свойства и методы класса Object

Как мы обсуждали ранее, все объекты в JavaScript наследуются от класса Object. Специализированные классы, такие как встроенный класс String или определенный пользователем класс Complex, определяют свои собственные свойства и методы, но кроме этого все объекты, независимо от своего класса, также поддерживают свойства и методы, определенные классом Object. По причине их универсальности эти свойства и методы представляют особый интерес.

8.7.1. Свойство constructor

Начиная с JavaScript 1.1 любой объект имеет свойство `constructor`, которое ссылается на функцию-конструктор, использованную для инициализации объекта. Например, если объект `o` создается с помощью конструктора `Complex()`, то свойство `o.constructor` ссылается на функцию `Complex`:

```
var o = new Complex(1,2);
o.constructor == Complex;  // Равно true
```

Каждый объект Complex (или объект любого другого типа), конечно, не имеет собственного уникального свойства `constructor`; вместо этого данное свойство наследуется от объекта-прототипа. Как говорилось ранее в этой главе, для каждой определяемой программистом функции JavaScript создает объект-прототип и присваивает его свойству `prototype` конструктора. Однако ранее мы не раскрывали тот факт, что объект-прототип изначально не пуст. При создании в нем имеется свойство `constructor`, ссылающееся на функцию-конструктор, то есть для любой функции `f` выражение `f.prototype.constructor` всегда равно `f` (если мы не установили его равным чему-то другому).

Функция-конструктор определяет класс объекта, поэтому свойство `constructor` может применяться для определения типа любого заданного объекта. Например, тип неизвестного объекта можно определить таким способом:

```
if ((typeof o == "object") && (o.constructor == Date))
    // Какие-то действия с объектом Date...
```

Однако существование свойства constructor не всегда гарантировано. Например, автор класса может заменить объект-прототип конструктора совершенно новым объектом, который не имеет свойства constructor.

8.7.2. Метод toString()

Метод toString() не требует аргументов; он возвращает строку, каким-либо образом представляющую тип и/или значение объекта, для которого он вызывается. JavaScript вызывает этот метод объекта во всех тех случаях, когда ему требуется преобразовать объект в строку. Например, это происходит, когда используется оператор + для конкатенации строк с объектом или при передаче объекта такому методу, как alert() или document.write().

Метод toString() по умолчанию не очень информативен. Например, следующие строки кода заставляют броузер вывести строку «[object Object]»:[1]

```
c = new Circle(1, 0, 0);
document.write(c);
```

Этот метод по умолчанию не отображает особенно полезной информации, поэтому многие классы определяют свои собственные версии метода toString(). Например, когда массив преобразуется в строку, мы получаем список элементов массива, каждый из которых преобразуется в строку, а когда в строку преобразуется функция, мы получаем исходный код этой функции.

Идея метода toString() состоит в том, что каждый класс объектов имеет свое собственное особое строковое представление, поэтому он должен определять соответствующий метод toString() для преобразования объектов в строковую форму. Поэтому, определяя класс, необходимо определить для него специальный метод toString(), чтобы экземпляры класса могли быть преобразованы в осмысленные строки. Строка должна содержать информацию о преобразуемом объекте, так как это полезно для отладочных целей. Если способ преобразования в строку выбран правильно, он также может быть полезным в самих программах.

В следующем коде показан метод toString(), который мы можем определить для класса Circle из примера 8.5:

```
Circle.prototype.toString = function () {
    return "[Круг радиуса " + this.r + ", с центром в ("
        + this.x + ", " + this.y + ").]";
}
```

[1] В клиентском JavaScript в Netscape, если атрибут language тега <script> явно установлен в «JavaScript 1.2», метод toString() ведет себя по-другому: он показывает имена и значения всех полей объекта, используя нотацию объектного литерала. Это нарушает спецификацию ECMAScript.

После определения такого метода `toString()` типичный объект Circle может быть преобразован в строку «Круг с радиусом, равным 1, с центром в точке (0,0)».

Вернувшись к примеру 8.6, мы увидим, что для нашего класса комплексных чисел Complex определяется метод `toString()`.

Интересная особенность метода `toString()`, определенного классом Object по умолчанию, состоит в том, что он открывает некоторую внутреннюю информацию о типе встроенных объектов. Этот метод `toString()` всегда возвращает строку в следующей форме:

```
[object класс]
```

Здесь *класс* – это внутренний тип объекта, и он обычно соответствует имени функции-конструктора этого объекта. Например, для объектов Array *класс* будет равен «Array», для объектов Function *класс* будет равен «Function», а для объектов Date *класс* будет равен «Date». Для встроенного объекта Math *класс* будет равен «Math», а всех объектов Error (в том числе экземпляров различных подклассов Error) – «Error». Для объектов клиентского JavaScript и любых других объектов, определенных реализацией JavaScript, *класс* будет величиной, определяемой реализацией (например, «Window», «Document» или «Form»). Для объектов, определяемых пользователем, таких как определенные выше в этой главе классы Circle и Complex, *класс* всегда равен «Object».

Обратите внимание, что значение *класс* предоставляет полезную информацию, которую нельзя получить с помощью оператора `typeof` (возвращающего для всех объектов либо «Object», либо «Function»). Значение *класс* предоставляет ту же информацию, что и описанное выше свойство `constructor`, но это значение предоставляется в форме строки, а не в форме функции-конструктора. Однако получить это значение можно только при помощи метода `toString()`, определенного по умолчанию в классе Object. Некоторые классы часто определяют собственные версии этого метода, поэтому мы не можем просто вызвать метод `toString()` объекта:

```
o.toString()  // Может вызвать специализированный метод toString() данного объекта
```

Вместо этого мы должны с помощью объекта `Object.prototype.toString` явно сослаться на функцию `toString()`, определенную по умолчанию, и обратиться к методу `apply()` этой функции, чтобы вызвать ее для требуемого объекта:

```
Object.prototype.toString.apply(o);  // Всегда вызывает toString(),
                                     // определенную по умолчанию
```

Этот прием позволяет определить функцию, предоставляющую расширенные возможности определения типа:

```
// Расширенная функция для определения типа. Возвращает строку, описывающую
// тип x. Обратите внимание, что для всех типов объектов, определенных
// пользователем, она возвращает "Object"
```

```
function Typeof(x) {
    // Начнем с оператора typeof
    var t = typeof x;
    // Если результат является информативным, возвращаем его
    if (t != "object")  return t;
    // В противном случае x - это объект. Получаем его класс, чтобы попытаться
    // узнать, что это за объект.
    var c = Object.prototype.toString.apply(x);  // Возвращает "[object class]"
    c = c.substring(8, c.length-1);              // Отбрасываем "[object" и "]"
    return c;
}
```

8.7.3. Метод toLocaleString()

В ECMAScript v3 и JavaScript 1.5 класс Object в дополнение к методу `toString()` определяет метод `toLocaleString()`. Назначение этого метода состоит в получении локализованного строкового представления объекта. Метод `toLocaleString()` по умолчанию, определяемый классом Object, сам не выполняет никакой локализации; он всегда возвращает в точности такую же строку, что и `toString()`. Однако подклассы могут определять собственные версии метода `toLocaleString()`. В ECMAScript v3 классы Array, Date и Number определяют методы `toLocaleString()`, которые возвращают локализованные значения.

8.7.4. Метод valueOf()

Метод `valueOf()` во многом похож на метод `toString()`, но вызывается, когда JavaScript требуется преобразовать объект в какое-либо элементарное значение, отличное от строки – обычно в число. Когда это возможно, функция должна возвращать элементарное значение, каким-либо образом представляющее значение объекта, на который ссылается ключевое слово `this`.

По определению объекты не являются элементарными значениями, поэтому большинство объектов не имеют эквивалентного элементарного типа. Поэтому метод `valueOf()`, определяемый по умолчанию классом Object, не выполняет преобразования и просто возвращает объект, с которым он был вызван. Такие классы, как Number и Boolean имеют очевидные элементарные эквиваленты, поэтому они переопределяют метод `valueOf()` так, чтобы он возвращал соответствующие значения. Именно поэтому объекты Number и Boolean могут вести себя во многом так же, как эквивалентные им элементарные значения.

Иногда можно определить класс, имеющий какой-то разумный элементарный эквивалент. В этом случае может потребоваться определить для этого класса специальный метод `valueOf()`. Если мы вернемся к примеру 8.6, то увидим, что метод `valueOf()` определен для класса Complex. Этот метод просто возвращает вещественную часть комплексного числа. Поэтому в числовом контексте объект Complex ведет себя так, как будто он является вещест-

венным числом без мнимой составляющей. Рассмотрим, например, следующий код:

```
var a = new Complex(5,4);
var b = new Complex(2,1);
var c = Complex.subtract(a,b);  // c это комплексное число {3,3}
var d = a - b;                  // d это число 3
```

При наличии метода valueOf() следует соблюдать одну предосторожность: в случае преобразования объекта в строку метод valueOf() иногда имеет приоритет перед методом toString(). Поэтому, когда для класса определен метод valueOf() и надо, чтобы объект этого класса был преобразован в строку, может потребоваться явно указать на это, вызвав метод toString(). Продолжим пример с классом Complex:

```
alert("c = " + c);             // Используется valueOf(); показывает "c = 3"
alert("c = " + c.toString());  // Показывает "c = {3,3}"
```

8.7.5. Метод hasOwnProperty()

Метод hasOwnProperty() возвращает true, если для объекта определено не унаследованное свойство с именем, указанным в единственном строковом аргументе метода. В противном случае он возвращает false. Например:

```
var o = new Object();
o.hasOwnProperty("undef");     // false: свойство не определено
o.hasOwnProperty("toString");  // false: toString - это унаследованное свойство
Math.hasOwnProperty("cos");    // true: объект Math имеет свойство cos
```

8.7.6. Метод propertyIsEnumerable()

Метод propertyIsEnumerable() возвращает true, если в объекте определено свойство с именем, указанным в единственном строковом аргументе метода, и это свойство будет перечислено циклом for/in. В противном случае метод возвращает false. Например:

```
var o = { x:1 };
o.propertyIsEnumerable("x");        // true: свойство существует
                                    // и является перечисляемым
o.propertyIsEnumerable("y");        // false: свойство не существует
o.propertyIsEnumerable("valueOf");  // false: свойство не является перечисляемым
```

Обратите внимание, спецификация ECMAScript утверждает, что propertyIsEnumerable() рассматривает только свойства, определенные непосредственно объектом, а не унаследованные свойства. Это неприятное ограничение делает данную функцию менее полезной, так как возвращаемое значение false может означать, что либо свойство не перечисляемое, либо оно перечисляемое, но наследуется от надкласса.

8.7.7. Метод isPrototypeOf()

Метод `isPrototypeOf()` возвращает `true`, если объект представляет собой объект-прототип аргумента. В противном случае метод возвращает `false`. Этот метод используется аналогично свойству `constructor`. Например:

```
var o = new Object();
Object.prototype.isPrototypeOf(o);         // true: o.constructor == Object
Object.isPrototypeOf(o);                   // false
o.isPrototypeOf(Object.prototype);         // false
Function.prototype.isPrototypeOf(Object);  // true: Object.constructor == Function
```

9

Массивы

В главе 8 были описаны объекты JavaScript – составной тип данных, содержащий именованные значения. В этой главе мы рассмотрим массивы – составной тип данных, содержащий пронумерованные значения. Обратите внимание, что массивы, которые мы будем обсуждать в этой главе, отличаются от ассоциативных массивов, описанных в предыдущей. Ассоциативные массивы сопоставляют значения со строками. Массивы, описанные в этой главе, – это обычные массивы с числовыми индексами; они связывают значения с неотрицательными целыми числами.

На протяжении всей книги мы часто рассматриваем объекты и массивы как отдельные типы данных. Это полезное и разумное упрощение – объекты и массивы можно рассматривать как разные типы для большинства задач программирования в JavaScript. Однако чтобы хорошо понять поведение объектов и массивов, следует знать правду: массив – это ни что иное, как объект с тонким слоем дополнительной функциональности. Это можно увидеть, применив оператор `typeof` к массиву, – будет получена строка «object». Обратите внимание, что дополнительная функциональность массивов была введена в JavaScript 1.1. В JavaScript 1.0 массивы не поддерживаются.

В этой главе описан базовый синтаксис работы с массивами, приемы программирования с использованием массивов и методы, работающие с массивами.

9.1. Массивы и элементы массива

Массив – это тип данных, содержащий или хранящий пронумерованные значения. Каждое пронумерованное значение называется *элементом* массива, а число, с которым связывается элемент, называется его *индексом*. Так как JavaScript – это нетипизированный язык, элемент массива может иметь

любой тип и различные элементы одного массива могут иметь разный тип. Элементы массива могут даже содержать другие массивы, что позволяет создавать массивы массивов.

9.1.1. Создание массивов

В JavaScript 1.1 и более поздних версиях массивы создаются с помощью конструктора `Array()` и оператора `new`. Конструктор `Array()` можно вызвать тремя различными способами.

Первый способ – вызов без аргументов:

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

Этот метод создает пустой массив без элементов.

Второй метод вызова конструктора `Array()` позволяет явно указать значения для первых *n* элементов массива:

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

В этом случае конструктор получает список аргументов. Каждый аргумент задает значение элемента и может иметь любой тип. Нумерация элементов массива начинается с 0. Свойство `length` массива устанавливается равным количеству элементов, переданных конструктору.

Третий способ вызова конструктора `Array()` – вызов с единственным числовым аргументом, задающим длину массива:

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

Эта форма позволяет создать массив с заданным количеством элементов (каждый из которых имеет значение `undefined`) и устанавливает свойство `length` массива равным указанному значению.[1]

И наконец, еще один способ создания массивов – это литералы. Литерал массива позволяет встраивать значение типа «массив» непосредственно в программу JavaScript так же, как мы определяем строковый литерал, помещая текст между символами кавычек. Для того чтобы создать литерал массива, просто поместите в текст список разделенных запятыми значений, заключенный в квадратные скобки. Например:

```
var primes = [2, 3, 5, 7, 11];
var a = ['a', true, 4.78];
```

[1] В клиентском JavaScript в Netscape если атрибут language тега <script> явно установлен в «JavaScript 1.2», эта третья форма конструктора Array() ведет себя так же, как вторая форма: создает массив с одним элементом и инициализирует этот элемент массива аргументом конструктора. Это не соответствует стандарту ECMAScript.

Литералы массивов могут содержать объектные литералы или литералы других массивов:

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

Подробное описание литералов массивов приведено в главе 3.

9.1.2. Чтение и запись элементов массива

Доступ к элементам массива осуществляется с помощью оператора `[]`. Слева от скобок должна присутствовать ссылка на массив. Внутри скобок должно находиться произвольное выражение, имеющее неотрицательное целое значение. Этот синтаксис пригоден как для чтения, так и для записи значения элемента массива. Следовательно, все приведенные ниже инструкции JavaScript допустимы:

```
value = a[0];
a[1] = 3.14;
i = 2;
a[i] = 3;
a[i + 1] = "hello";
a[a[i]] = a[0];
```

В некоторых языках первый элемент массива имеет индекс 1. Однако в JavaScript (как в C, C++ и Java) первый элемент массива имеет индекс 0.

Как мы видели в главе 8, оператор `[]` может также применяться для доступа к именованным свойствам объекта:

```
my['salary'] *= 2;
```

Это говорит нам о том, что объекты и массивы в фундаментальном смысле являются одним и тем же.

Обратите внимание, что индекс массива должен быть неотрицательным числом, меньшим $2^{32}-1$. Если число слишком большое, отрицательное или вещественное (или это логическое, объектное или другое значение), JavaScript преобразует его в строку и рассматривает результирующую строку как имя свойства объекта, а не как индекс массива. Таким образом, следующая строка создает новое свойство с именем «–1.23», а не новый элемент массива:

```
a[-1.23] = true;
```

9.1.3. Добавление новых элементов в массив

В таких языках, как C и Java, массив имеет фиксированное число элементов, которое должно быть задано при создании массива. Это не относится к JavaScript – в нем массив может иметь любое количество элементов, и это количество можно в любой момент изменить.

Для того чтобы добавить новый элемент в массив, достаточно присвоить ему значение:

```
a[10] = 10;
```

Массивы в JavaScript могут быть *разреженными*. Это значит, что индексы массива не обязательно попадают в последовательный диапазон чисел; реализация JavaScript может выделять память только для тех элементов массива, которые фактически в нем хранятся. Поэтому в результате выполнения следующих строк кода интерпретатор JavaScript скорее всего выделит память только для элементов массива с индексами 0 и 10 000, а не для 9 999 элементов, находящихся между ними:

```
a[0] = 1;
a[10000] = "это элемент 10,000";
```

Обратите внимание, что элементы массива также могут добавляться к объектам:

```
var c = new Circle(1,2,3);
c[0] = "это элемент массива в объекте!"
```

Этот пример просто определяет новое свойство объекта с именем «0». Однако добавление элемента массива в объект не делает объект массивом. Массивы, созданные с помощью конструктора `Array()` или литералов массива, имеют некоторые, описанные ниже особенности, которые не присущи объектам.

9.1.4. Длина массива

Все массивы, как созданные с помощью конструктора `Array()`, так и определенные с помощью литерала массива, имеют специальное свойство `length`, определяющее, сколько элементов содержится в массиве. Поскольку массивы могут иметь неопределенные элементы, более точная формулировка звучит так: свойство `length` *всегда* на единицу больше, чем самый большой номер элемента массива. В отличие от обычных свойств объектов, свойство `length` массива автоматически обновляется, чтобы оно оставалось корректным при добавлении новых элементов в массив. Следующий код это иллюстрирует:

```
var a = new Array();   // a.length == 0  (ни один элемент не определен)
a = new Array(10);     // a.length == 10 (определены пустые элементы 0-9)
a = new Array(1,2,3);  // a.length == 3  (определены элементы 0-2)
a = [4, 5];            // a.length == 2  (определены элементы 0 и 1)
a[5] = -1;             // a.length == 6  (определены элементы 0, 1 и 5)
a[49] = 0;             // a.length == 50 (определены элементы 0, 1, 5 и 49)
```

Помните, что индексы массива должны быть меньше $2^{32}-1$, то есть максимальное возможное значение свойства `length` равно $2^{32}-1$.

Наиболее распространенное применение свойства `length` – это организация цикла для перебора элементов массива:

```
var fruits = ["mango", "banana", "cherry", "pear"];
for(var i = 0; i < fruits.length; i++)
    alert(fruits[i]);
```

Конечно, в этом примере предполагается, что элементы массива расположены непрерывно и начинаются с элемента 0. Если это не так, следует перед обращением к элементу массива проверять, определен ли он:

```
for(var i = 0; i < fruits.length; i++)
    if (fruits[i] != undefined) alert(fruits[i]);
```

Свойство `length` массива доступно как для чтения, так и для записи. Если установить `length` меньшим его текущего значения, массив укорачивается до новой длины; любые элементы, более не попадающие в новый диапазон индексов, отбрасываются и их значения теряются. Если сделать `length` большим, чем его текущее значение, в конец массива добавляются новые, неопределенные элементы, увеличивая массив до нового размера.

Укорачивание массива путем установки его значения `length` – единственный способ реального укорачивания массива. Если удалить элемент массива при помощи оператора `delete`, то этот элемент станет неопределенным, но свойство `length` не изменится.

Обратите внимание, что хотя объектам могут быть присвоены элементы массива, объекты не имеют свойства `length`. Это свойство и его специальное поведение – наиболее важная особенность массивов. Другие особенности, отличающие массив от объектов, – это различные методы, определяемые классом Array, которые описаны в разделе 9.2 «Методы массивов».

9.1.5. Многомерные массивы

JavaScript не поддерживает истинно многомерные массивы, но позволяет неплохо иммитировать их при помощи массивов массивов. Для доступа к элементу данных в массиве массивов достаточно использовать оператор `[]` дважды. Предположим, что переменная `matrix` – это массив массивов чисел. Любой элемент `matrix[x]` – это массив чисел. Для доступа к определенному числу в массиве надо написать `matrix[x][y]`.

9.2. Методы массивов

Кроме оператора `[]`, с массивами можно работать посредством различных методов, предоставляемых классом Array. Эти методы представлены в следующих разделах. Многие из методов позаимствованы из языка программирования Perl; программистам, работавшим с Perl, они могут показаться знакомыми. Как обычно, здесь приведен только их обзор, а полные описания находятся в справочном разделе, посвященном базовому JavaScript.

9.2.1. join()

Метод `Array.join()` преобразует все элементы массива в строки и объединяет их. Можно указать необязательный строковый аргумент, предназначенный для разделения элементов в результирующей строке. Если разделитель не

задан, используется запятая. Например, следующие строки кода дают в результате строку «1,2,3»:

```
var a = [1, 2, 3];  // Создает новый массив с указанными тремя элементами
var s = a.join();   // s == "1,2,3"
```

В следующем коде задается необязательный разделитель, что дает несколько другой результат:

```
s = a.join(", ");   // s == "1, 2, 3"
```

Обратите внимание на пробел после запятой. Метод `Array.join()` является обратным по отношению к методу `String.split()`, создающему массив путем разбиения строки на фрагменты.

9.2.2. reverse()

Метод `Array.reverse()` меняет порядок элементов в массиве на противоположный и возвращает массив с переставленными элементами. Он делает это на месте, другими словами, – метод не создает новый массив с переупорядоченными элементами, а переупорядочивает их в уже существующем массиве. Например, следующий код, содержащий методы `reverse()` и `join()`, дает в результате строку «3,2,1»:

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

9.2.3. sort()

Метод `Array.sort()` на месте сортирует элементы массива и возвращает отсортированный массив. Если `sort()` вызывается без аргументов, то он сортирует элементы массива в алфавитном порядке (при необходимости временно преобразуя их в строки для выполнения сравнения):

```
var a = new Array("banana", "cherry", "apple");
a.sort();
var s = a.join(", ");  // s == "apple, banana, cherry"
```

Неопределенные элементы переносятся в конец массива.

Для сортировки в каком-либо порядке, отличном от алфавитного, можно передать методу `sort()` в качестве аргумента функцию сравнения. Эта функция принимает решение, какой из двух ее аргументов должен присутствовать раньше в отсортированном списке. Если первый аргумент должен предшествовать второму, функция сравнения должна вернуть отрицательное число. Если первый аргумент в отсортированном массиве должен следовать за вторым, то возвращаемое функцией число должно быть больше нуля. А если два значения эквивалентны (то есть порядок их расположения не важен), функ-

ция сравнения должна вернуть 0. Поэтому, например, для сортировки элемента в числовом порядке, а не в алфавитном, можно сделать следующее:

```
var a = [33, 4, 1111, 222];
a.sort();                // Алфавитный порядок:  1111, 222, 33, 4
a.sort(function(a,b) {   // Числовой порядок: 4, 33, 222, 1111
           return a-b;   // Возвращает значение < 0, 0, или > 0, в зависимости
       });               // от порядка сортировки a и b
```

Обратите внимание на удобное использование в этом коде функционального литерала. Функция сравнения вызывается только один раз, поэтому нет необходимости давать ей имя.

Как еще один пример сортировки элементов массива, вы можете выполнить алфавитную сортировку массива строк без учета регистра символов, передав методу функцию сравнения, преобразующую перед сравнением оба своих аргумента в нижний регистр (с помощью метода `toLowerCase()`). Вы можете придумать другие функции сортировки, сортирующие числа в различном экзотическом порядке: обратном числовом, нечетные числа перед четными и т. д. Более интересные возможности, конечно же, открываются, когда сравниваемые элементы массива представляют собой объекты, а не простые типы, такие как числа или строки.

9.2.4. concat()

Метод `Array.concat()` создает и возвращает новый массив, содержащий элементы исходного массива, для которого был вызван метод `concat()`, за которым следуют все аргументы, переданные `concat()`. Если какой-либо из этих аргументов сам является массивом, в результирующий массив будут добавлены его элементы. Однако обратите внимание, что рекурсивного разделения массивов массивов не происходит. Вот несколько примеров:

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

9.2.5. slice()

Метод `Array.slice()` возвращает *фрагмент*, или подмассив, указанного массива. Два аргумента метода задают начало и конец возвращаемого фрагмента. Возвращаемый массив содержит элемент, номер которого указан в качестве первого аргумента, и все последующие элементы до, но не включая элемент, номер которого указан во втором аргументе. Если указан только один аргумент, возвращаемый массив содержит все элементы от начальной позиции до конца массива. Если какой-либо из аргументов отрицателен, он задает номер элемента массива относительно конца массива. Так, аргумент,

равный –1, задает последний элемент массива, а аргумент, равный –3, задает третий элемент от последнего элемента массива. Вот несколько примеров:

```
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]
```

9.2.6. splice()

Метод `Array.splice()` — это универсальный метод для вставки или удаления элементов из массива. Он изменяет массив на месте, а не возвращает новый массив, как это делают `slice()` и `concat()`. Обратите внимание, что `splice()` и `slice()` имеют очень похожие имена, но выполняют различные операции.

Метод `splice()` может удалять элементы из массива, вставлять новые элементы в массив или выполнять обе операции одновременно. Элементы массива при необходимости смещаются, чтобы после вставки или удаления образовывалась непрерывная последовательность. Первый аргумент `splice()` задает позицию в массиве, с которой начинается вставка и/или удаление. Второй аргумент задает количество элементов, которые должны быть удалены (вырезаны) из массива. Если второй аргумент опущен, удаляются все элементы массива от начального до конца массива. Метод `splice()` возвращает массив удаленных элементов или (если ни один из элементов не был удален) пустой массив. Например:

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

Первые два аргумента `splice` задают элементы массива, подлежащие удалению. За этими аргументами может следовать любое количество дополнительных аргументов, задающих элементы, которые будут вставлены в массив, начиная с позиции, заданной первым аргументом. Например:

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

Обратите внимание, что, в отличие от `concat()`, `splice()` не разбивает на отдельные элементы вставляемые аргументы-массивы. То есть если методу передается массив для вставки, он вставляет сам массив, а не элементы этого массива.

9.2.7. push() и pop()

Методы `push()` и `pop()` позволяют нам работать с массивами как со стеками. Метод `push()` добавляет один или несколько новых элементов в конец масси-

ва и возвращает его новую длину.[1] Метод pop() делает обратное: удаляет последний элемент массива, уменьшает длину массива и возвращает удаленное им значение. Обратите внимание, что оба этих метода изменяют массив на месте, а не создают его модифицированную копию. Комбинация push() и pop() позволяет нам с помощью массива JavaScript реализовать стек «первым вошел – последним вышел». Например:

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

9.2.8. unshift() и shift()

Методы unshift() и shift() ведут себя во многом так же, как push() и pop(), за исключением того, что они вставляют и удаляют элементы в начало массива, а не в его конец. Метод unshift() смещает существующие элементы в сторону больших индексов для освобождения места, добавляет элемент или элементы в начало массива и возвращает новую длину массива. Метод shift() удаляет и возвращает первый элемент массива, смещая все последующие элементы вниз на одну позицию для занятия свободного места в начале массива. Например:

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

Обратите внимание на поведение unshift() при вызове с несколькими аргументами. Аргументы вставляются не по одному, а все сразу (как в случае применения метода splice()). Это значит, что в результирующем массиве они будут следовать в том же порядке, в каком они были указаны в списке аргументов. Будучи вставленными по одному, они бы расположились в обратном порядке.

---

[1] В Netscape, когда атрибут language тега <script> явно задан как «JavaScript 1.2», push() возвращает последнее значение, добавленное в массив, а не новую длину массива.

9.2.9. toString() и toLocaleString()

У массива, как и у любого другого объекта JavaScript, имеется метод toString(). Для массива этот метод преобразует каждый из его элементов в строку (вызывая в случае необходимости методы toString() для элементов массива) и выводит список этих строк через запятую. Обратите внимание, что результат не включает квадратных скобок или каких-либо других разделителей вокруг значения массива. Например:

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

Заметьте, что toString() возвращает ту же строку, что и метод join() при вызове его без аргументов.[1]

Метод toLocaleString() – это локализованная версия toString(). Она преобразует каждый элемент массива в строку, вызывая метод toLocaleString() элемента, а затем конкатенирует результирующие строки с использованием специфического для региона (и определенного реализацией) разделителя.

[1] В Netscape, когда атрибут language тега <script> задан как «JavaScript 1.2», toString() ведет себя более сложным образом: он преобразует массивы в строки, включающие квадратные скобки, и вставляет кавычки вокруг элементов массива, являющихся строками, так что результирующие строки представляют собой допустимые литералы массивов.

10

Регулярные выражения

Регулярное выражение – это объект, описывающий символьный шаблон. Класс RegExp в JavaScript представляет регулярные выражения, а объекты String и RegExp определяют методы, использующие регулярные выражения для выполнения поиска по маске и операций поиска/замены в тексте.[1]

Регулярные выражения JavaScript стандартизованы в ECMAScript v3. JavaScript 1.2 реализует подмножество регулярных выражений, требуемых стандартом ECMAScript v3, а полный стандарт реализован в JavaScript 1.5. Регулярные выражения JavaScript в значительной степени базируются на средствах регулярных выражений из языка программирования Perl. Грубо говоря, мы можем сказать, что JavaScript 1.2 реализует регулярные выражения Perl 4, а JavaScript 1.5 – большое подмножество регулярных выражений Perl 5.

Эта глава начинается с определения синтаксиса, посредством которого в регулярных выражениях описываются текстовые шаблоны. Затем мы перейдем к описанию тех методов классов String и RegExp, которые используют регулярные выражения.

10.1. Определение регулярных выражений

В JavaScript регулярные выражения представлены объектами RegExp. Объекты RegExp могут быть созданы посредством конструктора `RegExp()`, но ча-

---

[1] Регулярное выражение – это малопонятный термин, уходящий корнями в далекое прошлое. Синтаксис описания текстового шаблона действительно представляет собою особый тип выражений. Однако, как мы увидим, его синтаксис далек от регулярного! Регулярные выражения иногда называют «regexp» или просто «RE».

ще они создаются с помощью специального синтаксиса литералов. Так же, как строковые литералы задаются в виде символов, заключенных в кавычки, литералы регулярных выражений задаются в виде символов, заключенных в пару символов косой черты (/). Таким образом, код JavaScript может содержать строки, похожие на эту:

```
var pattern = /s$/;
```

Эта строка создает новый объект RegExp и присваивает его переменной `pattern`. Данный объект RegExp ищет любые строки, заканчивающиеся символом «s». (Скоро мы поговорим о грамматике определения шаблонов.) Это регулярное выражение также может быть определено с помощью конструктора `RegExp()`:

```
var pattern = new RegExp("s$");
```

Создание объекта RegExp – либо с помощью литерала, либо с помощью конструктора `RegExp()` – это самое простое. Более сложную задачу представляет собой описание нужного шаблона с помощью синтаксиса регулярных выражений. JavaScript понимает довольно полное подмножество синтаксиса регулярных выражений, используемых в Perl, поэтому если вы опытный Perl-программист, то уже знаете, как описывать шаблоны в JavaScript.

Спецификация шаблона регулярного выражения состоит из последовательности символов. Большинство символов, включая все алфавитно-цифровые, просто описывают символы, которые должны присутствовать буквально. Следовательно, регулярное выражение `/java/` ищет все строки, содержащие подстроку «java». Другие символы в регулярных выражениях не разыскиваются буквально, а имеют особое значение. Например, регулярное выражение `/s$/` содержит два символа. Первый символ, «s», обозначает буквальный поиск символа. Второй, «$» – это специальный метасимвол, обозначающий конец строки. Таким образом, это регулярное выражение соответствует любой строке, заканчивающейся символом «s».

В следующих разделах описаны различные символы и метасимволы, используемые в регулярных выражениях JavaScript. Однако обратите внимание, что полноценный учебник по грамматике регулярных выражений выходит за рамки этой книги. Полное описание синтаксиса можно найти в книгах по Perl, таких как «Programming Perl» Ларри Уолла (Larry Wall), Тома Кристиансена (Tom Christiansen) и Джона Орванта (Jon Orwant) (издательство O'Reilly).[1] Еще один отличный источник информации по регулярным выражениям – книга «Mastering Regular Expressions» Джеффри Фридла (Jeffrey E. F. Friedl) (O'Reilly).[2]

---

[1] Ларри Уолл, Том Кристиансен, Джон Орвант «Программирование на Perl». – Пер. с англ. – СПб: Символ-Плюс, 2002.

[2] Джеффри Фридл «Регулярные выражения». – Пер. с англ. – СПб: Питер, 2003.

10.1.1. Символы литералов

Как мы видели, все алфавитные символы и цифры в регулярных выражениях соответствуют сами себе. Синтаксис регулярных выражений JavaScript также поддерживает возможность указывать некоторые неалфавитные символы с помощью управляющих последовательностей, начинающихся с обратной косой черты (\). Например, последовательность \n соответствует символу новой строки. Эти символы перечислены в табл. 10.1.

Таблица 10.1. Символы литералов в регулярных выражениях

Символ	Соответствие
Алфавитно-цифровые символы	Самим себе
\0	Символ NUL (\u0000)
\t	Табуляция (\u0009)
\n	Перевод строки (\u000A)
\v	Вертикальная табуляция (\u000B)
\f	Перевод страницы (\u000C)
\r	Возврат каретки (\u000D)
\x*nn*	Символ из набора Latin, задаваемый шестнадцатеричным числом *nn*; например, \x0A – это то же самое, что \n
\u*xxxx*	Символ Unicode, заданный шестнадцатеричным числом *xxxx*; например, \u0009 – это то же самое, что \t
\c*X*	Управляющий символ ^*X*; например, \cJ эквивалентно символу перевода строки \n

Некоторые символы имеют в регулярных выражениях особый смысл. К ним относятся:

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

Их значение рассматривается в последующих разделах. Некоторые из них имеют специальный смысл только в определенном контексте регулярного выражения, а в других контекстах трактуются буквально. Однако, как правило, для того чтобы включить какой-либо из этих символов в регулярное выражение буквально, необходимо поместить перед ним символ \. Другие символы, такие как кавычки и @ (амперсанд), не имеют специального значения и просто соответствуют в регулярных выражениях самим себе.

Если вы не можете точно вспомнить, каким из символов должна предшествовать обратная косая черта, то можете спокойно помещать обратную косую черту перед любым из символов. Но обратите внимание, что многие буквы и цифры вместе с обратной косой чертой обретают специальное значение, поэтому любым буквам и цифрам, которые вы ищете буквально, не должна предшествовать обратная косая черта. Чтобы включить в регулярное выражение сам символ обратной косой черты, перед ним, очевидно, следует по-

местить другую косую черту. Например, следующее регулярное выражение соответствует любой строке, содержащей обратную косую черту: /\\/.

10.1.2. Классы символов

Отдельные символы литералов могут объединяться в *классы символов* путем помещения их в квадратные скобки. Класс символов соответствует любому символу, содержащемуся в этом классе. Следовательно, регулярное выражение /[abc]/ соответствует одному из символов «a», «b» или «c». Могут также определяться классы символов с отрицанием, соответствующие любому символу, кроме тех, которые указаны в скобках. Класс символов с отрицанием задается символом ^ в качестве первого символа, следующего за левой скобкой. Регулярное выражение /[^abc]/ соответствует любому символу, отличному от a, b или c. В классах символов диапазон символов может задаваться при помощи дефиса. Поиск всех символов латинского алфавита в нижнем регистре осуществляется посредством выражения /[a-z]/, а любую букву или цифру из набора символов Latin можно найти при помощи выражения /[a-zA-Z0-9]/.

Некоторые классы символов используются особенно часто, поэтому синтаксис регулярных выражений JavaScript включает специальные символы и управляющие (escape) последовательности для их обозначения. Так, \s соответствует символам пробела, табуляции и любым символам-разделителям Unicode (whitespaces), а \S – любым символам, *не* являющимся символами-разделителями Unicode. В табл. 10.2 приведена сводка этих спецсимволов и синтаксиса классов символов. (Обратите внимание, что некоторые из управляющих последовательностей классов символов соответствуют только символам ASCII, и не были расширены для работы с символами Unicode. Можно явно определить собственные классы символов Unicode, – например, /[\u0400-04FF]/ соответствует любому символу кириллицы.)

Таблица 10.2. Классы символов регулярных выражений

Символ	Соответствие
[...]	Один символ из указанных в скобках
[^...]	Один символ из тех, что не указаны в скобках
.	Любой символ, кроме перевода строки или другого разделителя строки Unicode
\w	Любой текстовый символ ASCII. Эквивалентно [a-zA-Z0-9_]
\W	Любой символ, не являющийся текстовым символом ASCII. Эквивалентно [^a-zA-Z0-9_]
\s	Любой символ-разделитель Unicode
\S	Любой символ, не являющийся символом-разделителем Unicode. Обратите внимание, что \w и \S – это не одно и то же
\d	Любые цифры ASCII. Эквивалентно [0-9]

Таблица 10.2 (продолжение)

Символ	Соответствие
\D	Любой символ, отличный от цифр ASCII. Эквивалентно [^0-9]
[\b]	Литерал символа «забой» (особый случай)

Обратите внимание, что управляющие последовательности специальных классов символов могут находиться в квадратных скобках. \s соответствует любому символу-разделителю, а \d соответствует любой цифре, следовательно /[\s\d]/ соответствует любому символу-разделителю или цифре. Обратите внимание на особый случай. Как мы увидим позднее, последовательность \b имеет особый смысл. Однако когда она используется в классе символов, то обозначает символ «забой». Поэтому для того чтобы обозначить символ «забой» в регулярном выражении буквально, используйте класс символов с одним элементом: /[\b]/.

10.1.3. Повторение

Изучив синтаксис регулярных выражений, мы можем описать числа из двух цифр (/\d\d/) и из четырех цифр (/\d\d\d\d/). Но мы не имеем способа, например, для описания числа, которое может состоять из любого количества цифр или строку из трех букв, за которыми следует необязательная цифра. Эти более сложные шаблоны используют синтаксис регулярных выражений, указывающий, сколько раз может повторяться данный элемент регулярного выражения.

Символы, обозначающие повторение, всегда следуют за шаблоном, к которому они применяются. Некоторые виды повторений используются довольно часто, и для обозначения этих случаев имеются специальные символы. Например, + соответствует одному или нескольким экземплярам предыдущего шаблона. В табл. 10.3 приведена сводка синтаксиса повторений. В следующих строках показано несколько примеров:

```
/\d{2,4}/     // Соответствует числу, содержащему от двух до четырех цифр
/\w{3}\d?/    // Соответствует в точности трем текстовым символам
              // и необязательной цифре
/\s+java\s+/  // Соответствует слову "java" с одним или несколькими пробелами до
              // и после него
/[^"]*/       // Соответствует нулю или более символам, отличным от кавычек
```

Таблица 10.3. Символы повторения в регулярных выражениях

Символ	Значение
{*n*,*m*}	Соответствует предшествующему шаблону, повторенному не менее *n*, но не более *m* раз
{*n*,}	Соответствует предшествующему шаблону, повторенному *n* или более раз
{*n*}	Соответствует в точности *n* экземплярам предшествующего шаблона

Символ	Значение
?	Соответствует нулю или одному экземпляру предшествующего шаблона (указанный шаблон) является необязательным. Эквивалентно {0,1}
+	Соответствует одному или более экземпляру предшествующего шаблона. Эквивалентно {1,}
*	Соответствует нулю или более экземплярам предшествующего шаблона. Эквивалентно {0,}

Будьте осторожны при использовании символов повторения * и ?. Они могут соответствовать отсутствию указанного перед ними шаблона и, следовательно, отсутствию символов. Например, регулярному выражению /a*/ соответствует строка «bbbb», поскольку в ней нет символа «a»!

10.1.3.1. «Нежадное» повторение

Символы повторения, перечисленные в табл. 10.3, соответствуют максимально возможному количеству повторений, при котором обеспечивается поиск последующих частей регулярного выражения. Мы говорим, что это «жадное» повторение. Кроме того, в JavaScript 1.5 и более поздних версиях (это одна из возможностей Perl 5, не реализованная в JavaScript 1.2) поддерживается повторение, выполняемое «нежадным» способом. Достаточно указать после символа или символов повторения вопросительный знак: ??, +?, *? или даже {1,5}?. Например, регулярное выражение /a+/ соответствует одному или более экземплярам буквы «a». Примененное к строке «aaa», оно соответствует всем трем буквам. При этом выражение /a+?/ соответствует одному или более экземплярам буквы «a» и выбирает наименьшее возможное число символов. Примененная к той же строке, эта маска соответствует только первой букве «a».

«Нежадное» повторение не всегда дает ожидаемый результат. Рассмотрим маску /a*b/, соответствующую нулю или более символов «a», за которыми следует символ «b». Примененная к строке «aaab», она соответствует целой строке. Теперь проверим «нежадную» версию: /a*?b/. Она должна соответствовать символу «b», перед которым следует наименьшее возможное количество букв «a». В случае применения к той же строке «aaab» можно ожидать соответствия лишь последнего символа «b». Однако на самом деле эта маска соответствует и всей строке, так же как ее «жадная» версия. Дело в том, что поиск по маске регулярного выражения выполняется путем нахождения первой позиции в строке, начиная с которой соответствие становится возможным. «Нежадная» версия нашей маски находит соответствие с первым символом строки, и именно это соответствие является окончательным; соответствие последующих символов даже не рассматривается.

10.1.4. Альтернативы, группировка и ссылки

Грамматика регулярных выражений включает специальные символы для задания альтернатив, подвыражений группировки и ссылок на предыдущие

подвыражения. Символ вертикальной черты | служит для разделения альтернатив. Например, /ab|cd|ef/ соответствует или строке «ab», или строке «cd», или строке «ef», а /\d{3}|[a-z]{4}/ – либо трем цифрам, либо четырем строчным буквам.

Обратите внимание, что альтернативы обрабатываются слева направо до тех пор, пока не будет найдено соответствие. Если левая альтернатива найдена, правая игнорируется, даже если может добиться «лучшего» соответствия. Поэтому когда к строке «ab» применяется шаблон /a|ab/, он будет соответствовать только первому символу.

Скобки имеют в регулярных выражениях несколько значений. Одно из них – это группировка отдельных элементов в одно подвыражение, так что элементы при использовании |, *, +, ? и прочих спецсимволов рассматриваются как одно целое. Например, /java(script)?/ соответствует слову «java», за которым следует необязательное «script», а /(ab|cd)+|ef)/ соответствует либо строке «ef», либо одному или более повторению одной из строк «ab» или «cd».

Другим применением скобок в регулярных выражениях является определение подмасок внутри шаблона. Когда в целевой строке найдено соответствие регулярному выражению, можно извлечь часть целевой строки, соответствующую любой конкретной подмаске, заключенной в скобки. (Мы увидим, как получить эти подстроки позднее в этой главе). Предположим, что требуется отыскать одну или более букв в нижнем регистре, за которыми следует одна или несколько цифр. Мы можем задать для этого шаблон /[a-z]+\d+/. Но предположим также, что нам нужны только цифры в конце каждого соответствия. Если мы поместим эту часть шаблона в скобки, то сможем извлечь цифры из любых найденных нами соответствий. Позднее мы объясним, как это делается.

С этим связано еще одно применение подвыражений в скобках, позволяющее делать ссылку назад к подвыражению из более поздней части того же регулярного выражения. Это достигается путем указания одной или нескольких цифр после символа \. Цифры ссылаются на позицию подвыражения в скобках внутри регулярного выражения. Например, \1 ссылается на первое подвыражение, а \3 - на третье. Обратите внимание, что поскольку подвыражения могут быть вложены одно в другое, при подсчете используется позиция левой скобки. Например, в следующем регулярном выражении ссылка на вложенное подвыражение ([Ss]cript) будет выглядеть как \2:

```
/([Jj]ava([Ss]cript)?)\sis\s(fun\w*)/
```

Ссылка на подвыражение регулярного выражения ссылается *не* на шаблон этого подвыражения, а на текст, соответствующий этому шаблону. Поэтому ссылки могут применяться для наложения ограничения, выбирающего части строки, содержащие точно такие же символы. Например, следующее регулярное выражение соответствует нулю или более символам внутри одинарных или двойных кавычек. Однако оно не требует, чтобы открывающие

и закрывающие кавычки соответствовали друг другу (то есть чтобы обе кавычки были одинарными или двойными):

```
/['"][^'"]*['"]/
```

Соответствия кавычек мы можем потребовать посредством такой ссылки:

```
/(['"])[^'"]*\1/
```

«\1» соответствует результату поиска в соответствии с первым подвыражением. В этом примере ссылка налагает ограничение, требующее, чтобы закрывающая кавычка соответствовала открывающей. Это регулярное выражение не допускает присутствия одинарных кавычек внутри двойных и наоборот. Недопустимо помещать ссылки внутрь классов символов, то есть мы не можем написать:

```
/(['"])[^\1]*\1/
```

Позднее в этой главе мы увидим, что этот вид ссылок на подвыражения представляет собой мощное средство для регулярных выражений, используемых в операциях поиска/замены.

В JavaScript 1.5 (но не в JavaScript 1.2) возможна группировка элементов в регулярном выражении без создания нумерованной ссылки на эти элементы. Вместо простой группировки элементов между (и) начните группу с (?: и закончите ее). Рассмотрим, например, следующую маску:

```
/([Jj]ava(?:[Ss]cript)?)\sis\s(fun\w*)/
```

Здесь подвыражение (?:[Ss]cript) нужно только для группировки, чтобы к группе мог быть применен символ повторения ?. Эти модифицированные скобки не создают ссылку, поэтому в данном регулярном выражении \2 ссылается на текст, соответствующий маске (fun\w*).

В табл. 10.4 приводится сводка операторов альтернативы, группировки и ссылки в регулярных выражениях.

Таблица 10.4. Символы альтернативы, группировки и ссылки в регулярных выражениях

Символ	Значение
|	Альтернативы. Соответствует либо подвыражению слева, либо подвыражению справа.
(...)	Группировка. Группирует элементы в единое целое, которое может использоваться с *, +, ?, | и т. п. Также запоминает символы, соответствующие этой группе для использования в последующих ссылках.
(?:...)	Только группировка. Группирует элементы в единое целое, но не запоминает символы, соответствующие этой группе.
n	Соответствует тем же символам, которые были найдены при первом соответствии группе с номером *n*. Группы – это подвыражения внутри скобок (возможно, вложенных). Номера группам присваиваются путем подсчета левых скобок слева направо. Группы, сформированные с помощью (?:, не нумеруются.

10.1.5. Задание позиции соответствия

Мы видели, что многие элементы регулярного выражения соответствуют одному символу в строке. Например, `\s` соответствует одному символу-разделителю. Другие элементы регулярных выражений соответствуют позициям в тексте, а не самим символам. Например, `\b` соответствует границе слова – границе между `\w` (текстовый символ ASCII) и `\W` (нетекстовый символ), или границе между текстовым символом ASCII и началом или концом строки.[1] Такие элементы, как `\b`, не задают каких-либо символов, которые должны присутствовать в найденной строке, однако они определяют допустимые позиции, в которых будет выполняться проверка соответствия. Иногда эти элементы называются якорными элементами регулярных выражений, так как они прикрепляют шаблон к определенной позиции в строке. Чаще других используются такие якорные элементы, как символ `^`, привязывающий шаблон к началу строки, и символ `$`, прикрепляющий шаблон к концу строки.

Например, слово «JavaScript», находящееся на отдельной строке, можно найти с помощью регулярного выражения `/^JavaScript$/`. Отдельное слово «Java» (а не префикс, например в «JavaScript») можно поискать по шаблону `/\sJava\s/`, который требует наличия пробела до и после слова. Но такое решение порождает две проблемы. Во-первых, оно найдет слово «Java», только если оно окружено пробелами с обеих сторон, и не сможет найти его в начале или в конце строки. Во-вторых, когда этот шаблон действительно найдет соответствие, возвращаемая им строка будет содержать ведущие и замыкающие пробелы, а это не совсем то, что нам нужно. Поэтому вместо шаблона для реальных символов-разделителей `\s` мы воспользуемся шаблоном (или якорем) для границ слова `\b`. Получится следующее выражение: `/\bJava\b/`. Элемент `\B` представляет собою якорь для позиции, не являющейся границей слова. То есть маска `/\B[Ss]cript/` соответствует «JavaScript» и «postscript», но не соответствует «script» или «Scripting».

В JavaScript 1.5 (но не в JavaScript 1.2) в качестве якорных условий могут также выступать произвольные регулярные выражения. Если поместить выражение между символами `(?=` и `)`, оно станет условием на последующие символы, требующим, чтобы следующие символы соответствовали указанному шаблону, но не включались в строку соответствия. Например, найти имя языка программирования JavaScript, но только там, где за ним следует двоеточие, можно посредством выражения `/[Jj]ava([Ss]cript)?(?=\:)/`. Эта маска находит слово «JavaScript» в «JavaScript: The Definitive Guide», но не находит слово «Java» в «Java in a Nutshell», так как после него нет двоеточия.

Если же ввести условие символами `(?!`, то это будет отрицательное условие на последующие символы, требующее, чтобы следующие символы не соответствовали указанному шаблону. Так, `/Java(?!Script)([A-Z]\w*)/` соответствует слову «Java», за которым следует большая буква и произвольное коли-

[1] За исключением класса символов (квадратных скобок), где `\b` соответствует символу «забой».

чество дополнительных текстовых символов ASCII, если только за «Java» не следует «Script». Эта маска соответствует «JavaBeans», но не соответствует «Javanese», соответствует «JavaScrip», но не соответствует «JavaScript» или «JavaScripter».

В табл. 10.5 приведена сводка якорных символов регулярных выражений.

Таблица 10.5. Якорные символы регулярных выражений

Символ	Значение
^	Поиск начала строкового выражения или, при многострочном поиске, начала строки.
$	Поиск конца строкового выражения, или, при многострочном поиске, конца строки.
\b	Поиск границы слова, то есть соответствует позиции между символом \w и символом \W или между символом \w и началом или концом строки. (Заметьте, однако, что [\b] соответствует символу забоя.)
\B	Поиск позиции, не являющейся границей слов.
(?=*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 следующие символы не соответствовали шаблону *p*.

10.1.6. Флаги

И еще один, последний элемент грамматики регулярных выражений. Флаги регулярных выражений задают высокоуровневые правила соответствия шаблонов. В отличие от остальной грамматики регулярных выражений, флаги указываются вне символов /; они указываются не между косыми чертами, а после второй из них. JavaScript 1.2 поддерживает два флага. Флаг i указывает, что поиск по шаблону должен быть нечувствительным к регистру. Флаг g указывает, что поиск должен быть глобальным – то есть, должны быть найдены все соответствия в строке. Оба флага могут быть объединены для выполнения глобального поиска, не чувствительного к регистру.

Например, для того чтобы выполнить безразличный к регистру поиск первого вхождения слова «java» (или «Java», «JAVA» и т. д.), можно воспользоваться нечувствительным к регистру регулярным выражением /\bjava\b/i. А для того чтобы найти все вхождения этого слова в строке, надо добавить флаг g: /\bjava\b/gi.

JavaScript 1.5 поддерживает дополнительный флаг m, который выполняет поиск по шаблону в многострочном режиме. Если строковое выражение, в котором выполняется поиск, содержит символы перевода строк, то в этом режиме якорные символы ^ и $, помимо того что они соответствуют началу и концу всего строкового выражения, также соответствуют началу и концу строки. Например, маска /Java$/im соответствует как «java», так и «Java\nis fun».

В табл. 10.6 приведена сводка флагов регулярных выражений. Заметим, что флаг g более подробно рассматривается далее в этой главе вместе с методами классов String и RegExp, используемых для фактического осуществления поиска.

Таблице 10.6. Флаги регулярных выражений

Символ	Значение
i	Выполняет поиск, нечувствительный к регистру.
g	Выполняет глобальный поиск, то есть находит все соответствия, а не останавливается после первого из них.
m	Многострочный режим. ^ соответствует началу строки или началу всего строкового выражения, а $ – концу строки или всего выражения.

10.1.7. Средства регулярных выражений Perl, не поддерживаемые в JavaScript

Мы говорили, что ECMAScript v3 определяет относительно полное подмножество средств регулярных выражений из Perl 5. Развитые средства Perl, не поддерживаемые ECMAScript, включают следующее:

- Флаги s (однострочный режим) и x (расширенный синтаксис)
- Управляющие последовательности \a, \e, \l, \u, \L, \U, \E, \Q, \A, \Z, \z и \G
- (?<= – положительное условие на предыдущие символы и (?<! – отрицательное условие на предыдущие символы
- Комментарий (?# и прочий расширенный синтаксис (?

10.2. Методы класса String для поиска по шаблону

До этого момента мы обсуждали грамматику создаваемых регулярных выражений, но мы не рассматривали, как эти регулярные выражения могут реально применяться в коде JavaScript. В данном разделе мы обсудим методы объекта String, в которых регулярные выражения привлекаются для выполнения поиска по шаблону и операций поиска-замены. А затем продолжим разговор о поиске по шаблону с регулярными выражениями JavaScript, рассмотрев объект RegExp, его методы и свойства. Обратите внимание, что последующее обсуждение – это лишь обзор различных методов и свойств, относящихся к регулярным выражениям. Полное описание можно найти в справочном разделе, посвященном базовому JavaScript.

Строки поддерживают четыре метода, опирающихся на регулярные выражения. Простейший из них – метод search(). Он принимает в качестве аргумента регулярное выражение и возвращает либо позицию символа в начале

первой найденной подстроки, либо –1, если соответствие не найдено. Например, следующий вызов возвращает 4:

```
"JavaScript".search(/script/i);
```

Если аргумент метода search() не является регулярным выражением, он сначала преобразуется путем передачи конструктору RegExp. Метод search() не поддерживает глобальный поиск и игнорирует флаг g в своем аргументе.

Метод replace() выполняет операцию поиска с заменой. Он принимает в качестве первого аргумента регулярное выражение, а в качестве второго – строку замены. Метод ищет в строке, для которой он вызван, соответствие указанному шаблону. Если регулярное выражение содержит флаг g, метод replace() заменяет все вхождения строки на строку замены, в противном случае он заменяет только первое найденное вхождение. Если первый аргумент replace() представляет собой не регулярное выражение, а строку, то метод выполняет буквальный поиск строки, а не преобразует его в регулярное выражение с помощью конструктора RegExp(), как это делает метод search(). В качестве примера мы можем воспользоваться методом replace() для выполнения единообразной расстановки заглавных букв в слове «JavaScript» для всей строки текста:

```
// Независимо от регистра символов заменяем словом с правильной капитализацией
text.replace(/javascript/gi, "JavaScript");
```

Однако replace() представляет собой более мощное средство, чем можно судить по этому примеру. Вспомните, что подвыражения в скобках, находящиеся внутри регулярного выражения, нумеруются слева направо, и что регулярное выражение запоминает текст, соответствующий каждому из подвыражений. Если в строке замены присутствует знак $ с цифрой, метод replace() заменяет эти два символа на текст, соответствующий указанному подвыражению. Это очень полезная возможность. Мы можем использовать ее, например, для замены прямых кавычек в строке на кавычки «66-99», которые имитируются ASCII-символами:

```
// Цитата – это кавычка, за которой следует любое число символов,
// отличных от кавычек (их мы запоминаем), за которыми следует
// еще одна кавычка
var quote = /"([^"]*)"/g;
// Заменяем прямые кавычки на типографские и оставляем без
// изменений содержимое цитаты, хранящееся в $1.
text.replace(quote, "``$1``");
```

Метод replace() предоставляет и другие ценные возможности, описанные на странице «String.replace()» в «Справочнике по базовому JavaScript». Самое важное, что следует отметить, – второй аргумент replace() может быть функцией, динамически вычисляющей строку замены.

Метод match() – это наиболее общий из методов класса String, опирающихся на регулярные выражения. Он принимает в качестве единственного аргумента регулярное выражение (или преобразует свой аргумент в регулярное вы-

ражение, передав его конструктору RegExp()) и возвращает массив, содержащий результаты поиска. Если в регулярном выражении установлен флаг g, метод возвращает массив всех соответствий, присутствующих в строке. Например:

```
"1 плюс 2 равно 3".match(/\d+/g)  // возвращает ["1", "2", "3"]
```

Если регулярное выражение не содержит флага g, метод match() не выполняет глобального поиска; он просто ищет первое соответствие. Однако match() возвращает массив, даже когда метод не выполняет глобального поиска. В этом случае первый элемент массива – это найденная строка, а все оставшиеся элементы представляют собой подвыражения регулярного выражения. Поэтому если match() возвращает массив a, a[0] будет содержать найденную строку целиком, a[1] – подстроку, соответствующую первому подвыражению и т. д. Проводя параллель с методом replace(), можно сказать, что в a[*n*] заносится содержимое $*n*.

Рассмотрим, например, разбор URL с помощью следующего кода:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Посетите мою домашнюю страницу по адресу http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];   // Содержит "http://www.isp.com/~david"
    var protocol = result[1];  // Содержит "http"
    var host = result[2];      // Содержит "www.isp.com"
    var path = result[3];      // Содержит "~david"
}
```

И наконец, имеется еще одна особенность метода match(), о которой вам следует знать. Возвращаемый им массив имеет, как и все массивы, свойство length. Однако когда match() вызывается с регулярным выражением без флага g, возвращаемый массив имеет еще два свойства: свойство index, содержащее номер позиции символа внутри строки, с которого начинается соответствие, и свойство input, являющееся копией строки, в которой выполнялся поиск. То есть в коде, приведенном выше, значение result.index будет равно 21, так как найденный URL начинается в тексте с 21-й позиции. Свойство result.input должно содержать ту же строку, что и переменная text. Для регулярного выражения r, в котором не установлен флаг g, вызов s.match(r) возвращает то же значение, что и r.exec(s). Немного позднее в этой главе мы обсудим метод RegExp.exec().

Последний из методов объекта String, в котором применяются регулярные выражения, – это split(). Этот метод разбивает строку, для которой он вызван, на массив подстрок, используя аргумент в качестве разделителя. Например:

```
"123,456,789".split(",");  // Возвращает ["123","456","789"]
```

Метод split() может также принимать в качестве аргумента регулярное выражение. Это делает метод более мощным. Например, мы можем указать

разделитель, допускающий произвольное число пробельных символов с обеих сторон:

```
"1,2, 3 , 4 ,5".split(/\s*,\s*/); // Возвращает ["1","2","3","4","5"]
```

Метод split() имеет и другие возможности. Полное описание приведено в разделе «String.split()» в части, посвященной базовому JavaScript.

10.3. Объект RegExp

Как было упомянуто в начале этой главы, регулярные выражения представлены в виде объектов RegExp. Кроме конструктора RegExp(), объекты RegExp поддерживают три метода и несколько свойств. Особенность класса RegExp состоит в том, что он определяет как свойства класса (или статические свойства), так и свойства экземпляра. То есть он определяет глобальные свойства, принадлежащие конструктору RegExp(), а также свойства, принадлежащие конкретным объектам RegExp. Методы поиска и свойства класса RegExp описаны в следующих двух разделах.

Конструктор RegExp() принимает один или два строковых аргумента и создает новый объект RegExp. Первый аргумент конструктора – это строка, содержащая тело регулярного выражения, то есть текст, который должен находиться между косыми чертами в литерале регулярного выражения. Обратите внимание, что и строковые литералы и регулярные выражения используют для управляющих последовательностей символ \, поэтому, передавая конструктору RegExp() регулярное выражение в виде строкового литерала, необходимо заменить все символы \ на \\. Второй аргумент RegExp() может отсутствовать. Если он указан, то задает флаги регулярного выражения. Это должен быть один из символов g, i, m либо комбинация этих символов. Например:

```
// Находит все пятизначные числа в строке. Обратите внимание на использование
// в этом примере символов \\.
var zipcode = new RegExp("\\d{5}", "g");
```

Конструктор RegExp() полезен, когда регулярное выражение создается динамически и поэтому не может быть представлено с помощью синтаксиса литералов регулярных выражений. Например, чтобы найти строку, введенную пользователем, надо создать регулярное выражение во время выполнения с помощью RegExp().

10.3.1. Методы класса RegExp для поиска по шаблону

Объекты RegExp определяют два метода, выполняющие операции поиска по шаблону; они ведут себя аналогично методам класса String, описанным ранее. Основной метод класса RegExp, применяемый для поиска по шаблону, – это exec(). Он похож на описанный выше метод match() класса String, за исключением того, что является методом класса RegExp, принимающим в качестве аргумента строку, а не методом класса String, принимающим аргу-

мент RegExp. Метод exec() исполняет регулярное выражение для указанной строки, то есть он ищет в строке соответствие. Если соответствие не найдено, метод возвращает null. Однако если соответствие найдено, он возвращает такой же массив, как массив, возвращаемый методом match() для поиска без флага g. Нулевой элемент массива содержит строку, соответствующую регулярному выражению, а все последующие элементы – подстроки, соответствующие всем подвыражениям. Кроме того, свойство index содержит номер позиции символа, которым начинается соответствующий фрагмент, а свойство input ссылается на строку, в которой выполнялся поиск.

В отличие от метода match(), exec() возвращает массив, структура которого не зависит от наличия в регулярном выражении флага g. Вспомните, что при передаче глобального регулярного выражения метод match() возвращает массив найденных соответствий. А exec() всегда возвращает одно соответствие, но предоставляет о нем полную информацию. Когда exec() вызывается для регулярного выражения, содержащего флаг g, метод устанавливает свойство lastIndex объекта регулярного выражения равным номеру позиции символа, следующего непосредственно за найденной подстрокой. Когда метод exec() вызывается для того же регулярного выражения второй раз, он начинает поиск с символа, позиция которого указана в свойстве lastIndex. Если exec() не находит соответствия, lastIndex равно 0. (Вы также можете установить lastIndex в ноль в любой момент, что следует делать во всех тех случаях, когда вы завершаете поиск до того, как нашли последнее соответствие в одной строке и начинаете поиск в другой строке с тем же объектом RegExp.) Это особое поведение позволяет нам вызывать exec() повторно для перебора всех соответствий регулярному выражению в строке. Например:

```
var pattern = /Java/g;
var text = "JavaScript - это более забавная штука, чем Java!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Найдено `" + result[0] + "'" +
        " в позиции " + result.index +
        "; следующий поиск начинается с " + pattern.lastIndex);
}
```

Еще один метод объекта RegExp – test(), который намного проще метода exc(). Он принимает строку и возвращает true, если строка соответствует регулярному выражению:

```
var pattern = /java/i;
pattern.test("JavaScript");  // Возвращает true
```

Вызов test() эквивалентен вызову exec(), возвращающему true, если exec() возвращает не null. По этой причине метод test() ведет себя так же, как метод exec() при вызове для глобального регулярного выражения: он начинает искать указанную строку с позиции, заданной свойством lastIndex, и если находит соответствие, устанавливает lastIndex равным номеру позиции символа, непосредственно следующего за найденным соответствием. Поэтому

мы можем сформировать с помощью метода `test()` цикл для обхода строки так же, как мы можем это сделать с помощью метода `exec()`.

Методы `search()`, `replace()` и `match()` класса String не используют свойство `lastIndex` как методы `exec()` и `test()`. На самом деле методы класса String просто сбрасывают `lastIndex` в 0. Если мы используем `exec()` или `test()` с шаблоном, в котором установлен флаг `g`, и выполняем поиск в нескольких строках, то мы должны либо найти все соответствия в каждой строке, чтобы свойство `lastIndex` автоматически сбросилось в ноль (это происходит, когда последний поиск оказывается неудачным), либо явно установить свойство `lastIndex` равным нулю самостоятельно. Если этого не сделать, то поиск в новой строке может начаться с некоторой произвольной позиции, а не с начала. И наконец, помните, что особое поведение свойства `lastIndex` относится только к регулярным выражениям с флагом `g`. Методы `exec()` и `test()` игнорируют свойство `lastIndex` объектов RegExp, в которых отсутствует флаг `g`.

10.3.2. Свойства экземпляра RegExp

Каждый объект RegExp имеет пять свойств. Свойство `source` — это доступная только для чтения строка, содержащая текст регулярного выражения. Свойство `global` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `g`. Свойство `ignoreCase` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `i`. Свойство `multiline` — это доступное только для чтения логическое значение, определяющее, имеется ли в регулярном выражении флаг `m`. И последнее свойство `lastIndex` — это целое число, доступное для чтения и записи. Для масок с флагом `g` это свойство содержит номер позиции в строке, с которой должен быть начат следующий поиск. Как описано в предыдущем разделе, оно используется методами `exec()` и `test()`.

11

Прочие вопросы программирования на JavaScript

Здесь рассматриваются разнородные вопросы программирования на JavaScript, которые, будь они рассмотрены в предыдущих главах, сделали бы их непроходимыми. Теперь же, разобравшись в базовом языке JavaScript, вы готовы взяться за более сложные и детально освещенные темы, представленные здесь. Однако при желании можно перейти к другому материалу, изучить специфику клиентского JavaScript, а к этой главе вернуться позднее.

11.1. Преобразования типов данных

Мы видели, что JavaScript – нетипизированный язык (точнее, слабо типизированный или динамически типизированный). Это значит, что мы не должны задавать тип переменной при ее объявлении. Нетипизированность придает JavaScript гибкость и простоту, предпочтительную для языка сценариев (хотя эти особенности достаются ценой отсутствия строгости, важной для длинных и более сложных программ, которые часто пишутся на более строгих языках, таких как C или Java). Важной чертой гибкой работы с типами данных в JavaScript являются выполняемые интерпретатором автоматические преобразования данных. Например, если методу `document.write()` передается число, JavaScript автоматически преобразует его в эквивалентное строковое представление. Аналогично, проверяя строковое значение в условии инструкции `if`, JavaScript автоматически преобразует эту строку в логическое значение – в `false`, если строка пуста, и в `true` в противном случае.

Основное правило состоит в том, что когда значение одного тип встречается в контексте, где требуется значение какого-то другого типа, JavaScript автоматически пытается преобразовать значение требуемым образом. Поэтому, например, в контексте, предполагающем логическую переменную, число преобразуется в логическое значение. Объект в строковом контексте преоб-

разуется в строку. Строку в числовом контексте JavaScript пытается преобразовать в число. В табл. 11.1 приведены все эти преобразования, выполняемые над конкретным типом данных в конкретном контексте. В следующих разделах преобразования типов в JavaScript описываются более подробно.

Таблица 11.1. Автоматические преобразования типов данных

	Контекст			
Значение	**Строковый**	**Числовой**	**Логический**	**Объектный**
Неопределенное значение	"undefined"	NaN	false	Ошибка
Null	"null"	0	false	Ошибка
Непустая строка	Как есть	Числовое значение строки или NaN	true	Объект String
Пустая строка	Как есть	0	false	Объект String
0	"0"	Как есть	false	Объект Number
NaN	"NaN"	Как есть	false	Объект Number
Бесконечность	"Infinity"	Как есть	true	Объект Number
Минус бесконечность	"-Infinity"	Как есть	true	Объект Number
Любое другое число	Строковое значение числа	Как есть	true	Объект Number
True	"true"	1	Как есть	Объект Boolean
False	"false"	0	Как есть	Объект Boolean
Объект	toString()	valueOf() или toString(), или NaN	true	Как есть

11.1.1. Преобразования «объект–элементарный тип»

В табл. 11.1 показано, как объекты JavaScript преобразуются в элементарные значения. Однако некоторые детали этого преобразования требуют дополнительного пояснения. Во-первых, ненулевой объект (не равный null) в логическом контексте преобразуется в true. Это справедливо для всех объектов (в том числе для всех массивов и функций), даже для объектов-оберток, обозначающих элементарные значения, преобразуемых в false. Например, в логическом контексте все приведенные ниже объекты преобразуются в true:[1]

[1] Однако обратите внимание, что в JavaScript 1.1 и 1.2 все эти объекты преобразуются в false, что соответствует спецификации ECMAScript.

```
new Boolean(false)  // Внутреннее значение равно false, но объект
                    // преобразуется в true
new Number(0)
new String("")
new Array()
```

В табл. 11.1 показано, что преобразование объектов в числа начинается с вызова метода `valueOf()` этого объекта. Большинство объектов наследуют стандартный метод `valueOf()` класса Object, который просто возвращает сам объект. Так как стандартный метод `valueOf()` не возвращает элементарного значения, JavaScript затем пытается преобразовать объект в число путем вызова его метода `toString()` и преобразования полученной строки в число.

Для массивов это приводит к интересным результатам. Вспомните, что метод `toString()`преобразует элементы массива в строки, а затем возвращает результат конкатенации этих строк с запятыми между ними. Следовательно, массив без элементов преобразуется в пустую строку, которая (как вы видели в табл. 11.1) преобразуется в 0! Кроме того, если массив состоит из одного элемента, числа *n*, то массив преобразуется в строковое представление этого числа, которое затем преобразуется обратно в само число *n*. Если массив содержит более одного элемента, или если его единственный элемент не является числом, массив преобразуется в `NaN`.[1]

В табл. 11.1 указано, как преобразуется объект в строковом контексте, и как он преобразуется в числовом контексте. Однако в JavaScript контекст не всегда определяется однозначно! Оператор + и операторы сравнения (<, <=, > и >=) работают как с числами, так и со строками, поэтому когда объект используется с одним из этих операторов, неясно, во что он должен преобразовываться – в число или в строку. В большинстве случаев JavaScript сначала пытается преобразовать объект путем вызова метода `valueOf()`. Если этот метод возвращает элементарное значение (обычно число), используется это значение. Однако часто `valueOf()` просто возвращает объект без преобразования; в этом случае JavaScript затем пытается преобразовать объект в строку с помощью вызова метода `toString()`.

Имеется только одно исключение из этого правила преобразования: когда объект Date используется с оператором +, преобразование выполняется с помощью метода `toString()`. Существование этого исключения обусловлено тем, что Date имеет как метод `toString()`, так и метод `valueOf()`. При использовании Date с оператором + почти всегда требуется выполнить строковую конкатенацию. Но когда Date участвует в операциях сравнения, почти всегда требуется выполнить числовое сравнение, для того чтобы определить, какой из двух моментов времени предшествовал другому.

Большинство объектов либо не имеют метода `valueOf()`, либо этот метод не возвращает полезных результатов. Когда к объекту применяется оператор +, обычно имеет место строковая конкатенация, а не сложение. Когда к объек-

---

[1] Однако заметим, что в JavaScript 1.1 и 1.2 массив в числовом контексте преобразуется в значение его длины.

ту применяется оператор сравнения, обычно выполняется строковое сравнение, а не числовое.

Объект, определяющий специальный метод `valueOf()`, может вести себя по-другому. Определив метод `valueOf()`, возвращающий число, можно применять к своему объекту арифметические и другие операторы, но добавление своего объекта к строке будет выполняться не так, как ожидается: метод `toString()` больше не вызывается и в конкатенации участвует строковое представление числа, возвращаемого методом `valueOf()`.

И наконец, помните, что метод `valueOf()` не называется `toNumber()`, строго говоря, его задача состоит в преобразовании объекта в осмысленное элементарное значение, поэтому некоторые объекты могут иметь методы `valueOf()`, возвращающие строки.

11.1.2. Явные преобразования типов

В табл. 11.1 перечислены автоматические преобразования типов, выполняемые JavaScript. Также поддерживается явное преобразование типов. JavaScript не определяет оператор преобразования, как в языках C, C++ и Java, но предоставляет сходные средства.

В JavaScript 1.1 (и в стандарте ECMA-262) `Number()`, `Boolean()`, `String()` и `Object()` могут вызываться не только как конструкторы, но и как функции. Будучи вызваны таким образом, эти функции пытаются преобразовать свои аргументы в соответствующий тип. Например, вы можете преобразовать любое значение `x` в строку с помощью `String(x)` и любое значение `y` в объект с помощью `Object(y)`.

Имеется еще несколько приемов, которые могут быть полезны для выполнения явных преобразований типов. Для того чтобы преобразовать значение в строку, выполните его конкатенацию с пустой строкой:

```
var x_as_string = x + "";
```

Чтобы преобразовать значение в число, вычтите из него ноль:

```
var x_as_number = x - 0;
```

Принудительно преобразовать какое-либо значение в логическое можно с помощью оператора `!`, примененного дважды:

```
var x_as_boolean = !!x;
```

Из-за склонности JavaScript к автоматическому преобразованию данных в необходимый в данный момент тип, явные преобразования обычно не требуются. Однако изредка они оказываются полезными, и могут применяться для того, чтобы сделать программу понятнее и точнее.

11.1.3. Преобразование чисел в строки

Преобразование чисел в строки выполняется в JavaScript, возможно, чаще других. Хотя обычно это происходит автоматически, имеется несколько полезных способов явного преобразования этого рода. Два мы уже видели:

```
var string_value = String(number);  // Использование конструктора String()
                                    // в качестве функции
var string_value = number + "";      // Конкатенация с пустой строкой
```

Еще одну возможность предоставляет метод `toString()`:

```
string_value = number.toString();
```

Метод `toString()` объекта Number (элементарные числовые значения преобразуются в объекты Number, поэтому вы можете вызвать этот метод) принимает необязательный аргумент, задающий основание системы счисления, в которую будет выполнено преобразование. Если аргумент не указан, преобразование выполняется по основанию 10. Но можно преобразовывать числа и с другими основаниями (между 2 и 36).[1] Например:

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

Недостаток версий JavaScript до версии 1.5 состоит в отсутствии стандартного способа, позволяющего указать количество десятичных знаков в строке, получаемой в результате преобразования числа, или потребовать использования экспоненциальной нотации. Это может усложнить отображение чисел, имеющих традиционные форматы, например денежных значений.

ECMAScript v3 и JavaScript 1.5 обходят это препятствие, добавляя в класс Number три новых метода преобразования числа в строку. Метод `toFixed()` преобразует число в строку и выводит указанное количество цифр после десятичной точки, не используя экспоненциальной нотации. Метод `toExponential()` преобразует число в строку, записывая его в экспоненциальной нотации с одной цифрой перед десятичной точкой и указанным количеством цифр после нее. Метод `toPrecision()` выводит число с помощью указанного количества значащих цифр. Если такого количества значащих цифр недостаточно для вывода целой части числа, оно записывается в экспоненциальной нотации. Обратите внимание, что все три метода корректно округляют

---

[1] Обратите внимание, что спецификация ECMAScript поддерживает указание основания системы счисления для метода `toString()`, но допускает, что для любого отличного от 10 основания метод возвращает строку, зависящую от реализации. Следовательно, совместимые реализации могут просто игнорировать аргумент и всегда возвращать результат по основанию 10. Но реализации от Netscape и Microsoft этот аргумент учитывают.

финальные цифры результирующей строки. Посмотрите на следующие примеры:

```
var n = 123456.789;
n.toFixed(0);          // "123457"
n.toFixed(2);          // "123456.79"
n.toExponential(1);    // "1.2e+5"
n.toExponential(3);    // "1.235e+5"
n.toPrecision(4);      // "1.235e+5"
n.toPrecision(7);      // "123456.8"
```

11.1.4. Преобразование строк в числа

Мы видели, что в числовом контексте строки, представляющие числа, автоматически преобразуются в реальные числа. Как было показано выше, это преобразование можно сделать явным:

```
var number = Number(string_value);
var number = string_value - 0;
```

Такое преобразование неудобно из-за своей излишней строгости. Оно работает только с десятичными числами, и хотя преобразование допускает ведущие и замыкающие пробелы, оно не допускает наличия в строке каких-либо непробельных символов после числа.

Для более гибких преобразований можно применять функции `parseInt()` и `parseFloat()`. Эти функции преобразуют и возвращают любое число, присутствующее в начале строки, игнорируя любые замыкающие нечисловые символы. Функция `parseInt()` обрабатывает только целые, а `parseFloat()` — как целые числа, так и числа с плавающей точкой. Если строка начинается с «0x» или «0X», `parseInt()` интерпретирует ее как шестнадцатеричное число.[1] Например:

```
parseInt("3 blind mice");    // Возвращает 3
parseFloat("3.14 meters");   // Возвращает 3.14
parseInt("12.34");           // Возвращает 12
parseInt("0xFF");            // Возвращает 255
```

Функция `parseInt()` может иметь второй аргумент, указывающий основание системы счисления обрабатываемого числа. Допустимые значения — от 2 до 36. Например:

```
parseInt("11", 2);           // Возвращает 3 (1*2 + 1)
parseInt("ff", 16);          // Возвращает 255 (15*16 + 15)
parseInt("zz", 36);          // Возвращает 1295 (35*36 + 35)
```

[1] Спецификация ECMAScript говорит о том, что если строка начинается «0» (но не «0x» или «0X»), `parseInt()` может анализировать его либо как восьмеричное число, либо как десятичное. Так как поведение не определено, никогда не используйте `parseInt()` для обработки чисел с ведущими нулями, если только не указываете основание явно!

```
parseInt("077", 8);          // Возвращает 63 (7*8 + 7)
parseInt("077", 10);         // Возвращает 77 (7*10 + 7)
```

Если `parseInt()` и `parseFloat()` не могут преобразовать указанную строку в число, они возвращают `NaN`.

```
parseInt("eleven");          // Возвращает NaN
parseFloat("$72.47");        // Возвращает NaN
```

11.2. Передача по значению и по ссылке

В JavaScript, как во всех языках программирования, имеется три основных способа манипуляции со значением. Во-первых, его можно скопировать, например, присвоив его новой переменной. Во-вторых, можно передать его в качестве аргумента функции или методу. В-третьих, сравнить его с другим значением, чтобы увидеть, равны ли они. Чтобы понимать язык программирования, вы должны понимать, как в нем выполняют эти три операции.

Для работы с данными имеются две фундаментально различных технологии. Эти две технологии называются «передача по значению» и «передача по ссылке». В случае передачи по значению, важны именно *данные*. Присваивание создает копию реального значения, и эта копия сохраняется в переменной, свойстве объекта или элементе массива; копия и оригинал – это два совершенно независимых значения, хранящихся отдельно. Когда данные передаются функции по значению, она получает копию данных; если функция изменяет значение, изменение влияет только на копию данных, принадлежащую функции, не затрагивая исходные данные. И наконец, когда данные сравниваются по значению с другими данными, два различных фрагмента данных должны представлять в точности то же значение (обычно они оказываются равными при побайтовом сравнении).

Еще одна технология работы со значением – по ссылке. В этом случае имеется только одна реальная копия значения; манипуляции выполняются со ссылками на это значение.[1] Если манипуляции со значением выполняются по ссылке, переменные не содержат этого значения непосредственно; в них хранятся только ссылки на него. Именно ссылки копируются, передаются и сравниваются. Поэтому в присваивании, выполненном по ссылке, присваивается именно ссылка на значение, а не копия самого значения. После присваивания новая переменная ссылается на то же значение, на которое ссылается исходная переменная. Обе ссылки одинаково корректны и обе могут использоваться для операций со значением, – если значение изменяется по одной ссылке, это изменение также видно и по второй ссылке. То же самое происходит при передаче по ссылке значения в функцию. Функции передается ссылка на значение, и функция может использовать эту ссылку для из-

[1] Тем, кто программирует на C, и всем остальным, кто знаком с указателями, должен быть понятен термин «ссылка» в этом контексте. Однако обратите внимание, что JavaScript не поддерживает указатели.

менения значения; любые такие изменения видны вне функции. И наконец, когда значение сравнивается по ссылке с другим значением, сравниваются две ссылки, чтобы проверить, ссылаются ли они на одну и ту же копию значения; ссылки на два различных значения, оказавшихся эквивалентными (т. е. состоящими из одних и тех же байтов) не рассматриваются как равные.

Это две очень разные технологии работы со значениями, и они имеют важные нюансы, которые необходимо понимать (см. табл. 11.2). Это обсуждение работы с данными по значению и по ссылке является общим, но различия двух технологий относятся ко всем языкам программирования. В последующих разделах объясняется, как эти различия применяются к JavaScript, какие операции и с какими типами данных выполняются по значению, а какие – по ссылке.

Таблица 11.2. Передача по значению и по ссылке

Действие	По значению	По ссылке
Копирование	Значение на самом деле копируется; получаются две различных, независимых копии.	Копируется только ссылка на значение. Если значение изменяется через новую ссылку, это изменение также становится видимым через исходную ссылку.
Передача параметра	Функции передается отдельная копия значения; изменения в ней не видны за пределами функции.	Функции передается ссылка на значение. Если функция изменяет значение через переданную ссылку, изменения становятся видимыми за пределами функции.
Сравнение	Чтобы увидеть, что значения совпадают, сравниваются два различных значения (часто побайтно).	Сравниваются ссылки на предмет проверки, ссылаются ли они на одно и то же значение. Две ссылки на отдельные значения не равны, даже если два значения состоят из одних и тех же байтов.

11.2.1. Элементарные и ссылочные типы

Основное правило JavaScript гласит: манипуляции с элементарными типами выполняются по значению, а со ссылочными типами, как следует из их названия, – по ссылке. Числа и логические значения – это элементарные типы JavaScript, потому что они не содержат ничего, кроме небольшого фиксированного количества байтов, которые могут легко обрабатываться на низких уровнях интерпретатора JavaScript. Объекты являются ссылочными типами. Массивы и функции представляют собой специальные типы объектов и, следовательно, также являются ссылочными типами. Эти типы данных могут содержать произвольное количество свойств или элементов, поэтому с ними нельзя работать так же легко, как с элементарными значениями. Значения типа «объект» или «массив» могут быть довольно большими, поэтому нет смысла манипулировать этими данными по значению, так

как это может привести к неэффективному копированию и сравнению больших объемов данных.

А как же строки? Строки могут иметь произвольную длину, поэтому может показаться, что строки должны быть ссылочным типом. Тем не менее в JavaScript они обычно рассматриваются как элементарные типы просто потому, что не являются объектами. Строки на самом деле не укладываются в деление на элементарные и ссылочные типы. О строках и их поведении у нас будет что сказать позднее.

Лучший способ понять различия между работой с данными по значению и по ссылке – разобрать примеры. Изучите их внимательно, обращая внимание на комментарии. Пример 11.1 выполняет копирование, передачу параметров и сравнение чисел. Так как числа являются элементарным типом, этот пример иллюстрирует манипуляции с данными по значению.

Пример 11.1. Копирование, передача параметров и сравнение по значению

```
// Сначала мы проиллюстрируем копирование по значению
var n = 1;  // В переменной n содержится значение 1
var m = n;  // Копирование по значению: переменная m содержит отдельное
            // значение 1

// Это функция, которую мы будем использовать для иллюстрации передачи
// по значению. Как мы увидим, функция не работает так, как нам бы
// хотелось.
function add_to_total(total, x)
{
    total = total + x;  // Это строка изменяет только внутреннюю копию
                        // переменной total
}

// Теперь вызовем функцию, передав ей по значению числа, содержащиеся в
// переменных n и m. Значение n копируется, и это скопированное значение
// получает внутри функции имя total. Функция прибавляет копию m к
// данной копии n. Но прибавление чего-либо к копии n не влияет на исходное
// значение n вне функции. Вот почему вызов этой функции оказывается
// бесполезным.
add_to_total(n, m);

// Теперь посмотрим на сравнение по значению.
// В следующей строке кода литерал 1 это совершенно отдельное числовое
// значение, закодированное непосредственно в программе. Мы сравниваем его
// со значением, хранящимся в переменной n. При сравнении по значению байты
// двух чисел проверяются на совпадение.
if (n == 1) m = 2;  // n содержит то же самое значение, что и литерал 1,
                    // m теперь равно 2.
```

Теперь рассмотрим пример 11.2. Этот пример копирует, передает в качестве параметра и сравнивает объект. Так как объекты – это ссылочный тип, эти манипуляции выполняются по ссылке. В данном примере используются

объекты Date, более подробно рассмотренные в справочном разделе, посвященном базовому JavaScript.

Пример 11.2. Копирование, передача параметров и сравнение по ссылке

```
// Здесь мы создаем объект, представляющий дату Рождества 2001 года.
// Переменная xmas содержит ссылку на объект, а не сам объект
var xmas = new Date(2001, 11, 25);

// При копировании по ссылке мы получаем нувую ссылку на исходный объект
var solstice = xmas;  // Обе переменные теперь ссылаются на один и тот же объект

// Здесь мы изменяем объект через новую ссылку на него
solstice.setDate(21);

// Это изменение видно и через исходную ссылку
xmas.getDate();  // Возвращает 21, а не исходное значение 25

// То же самое происходит, когда в функции передаются объекты и массивы.
// Следующая функция прибавляет значение к каждому элементу массива.
// Функции передается ссылка на массив, а не его копия. Поэтому функция может
// изменить содержимое массива через ссылку, и эти изменения будут видимы
// после завершения работы функции.
function add_to_totals(totals, x)
{
    totals[0] = totals[0] + x;
    totals[1] = totals[1] + x;
    totals[2] = totals[2] + x;
}

// И наконец, мы рассмотрим сравнение по ссылке.
// Когда мы сравним две переменные, определенные выше, мы обнаружим, что
// они равны, так как ссылаются на один и тот же объект, даже несмотря на то,
// что мы пытались сделать так, чтобы они ссылались на разные даты.
(xmas == solstice)  // Равно true

// Две переменные, определенные ниже, ссылаются на два различных объекта,
// которые оба представляют одну и ту же дату.
var xmas = new Date(2001, 11, 25);
var solstice_plus_4 = new Date(2001, 11, 25);

// Но по правилам "сравнения по ссылке" различные объекты не являются равными!
(xmas != solstice_plus_4)  // Равно true
```

Перед тем как мы завершим тему работы с объектами и массивами по ссылке, следует прояснить вопрос терминологии. Фраза «передача по ссылке» может иметь разные значения. Для некоторых читателей эта фраза относится к способу вызова функции, позволяющему присваивать новые значения ее аргументам и делать эти измененные значения доступными вне функции. Это не тот смысл, с которым этот термин используется в книге. Здесь мы имеем в виду, что функции передается именно ссылка, а не сам объект.

Функция может использовать ссылку для изменения свойства объекта или элементов массива. Но если функция переписывает ссылку на массив ссылкой на новый объект или массив, это изменение не будет видимо вне функции. Читатели, знакомые с другим смыслом этого термина, могут предпочесть называть это передачей объектов и массивов по значению, но передаваемое значение фактически представляет собой ссылку, а не сам объект. Пример 11.3 иллюстрирует это.

Пример 11.3. Сами ссылки передаются по значению

```
// Это еще одна версия функции add_to_totals(). Она, однако, не работает,
// так как вместо изменения самого массива она пытается изменить ссылку
// на массив.
function add_to_totals2(totals, x)
{
    newtotals = new Array(3);
    newtotals[0] = totals[0] + x;
    newtotals[1] = totals[1] + x;
    newtotals[2] = totals[2] + x;
    totals = newtotals;  // Эта строка не имеет влияния вне функции
}
```

11.2.2. Копирование и передача строк

Как было упомянуто выше, строки JavaScript не укладываются в деление на элементарные и ссылочные типы. Так как строки – это не объекты, естественно предположить, что они представляют собой элементарные типы. А если они являются элементарными типами, то по правилам, приведенным выше, работа с ними выполняется по значению. Но поскольку строки могут иметь произвольную длину, их побайтное копирование, передача и сравнение могут показаться неэффективными. Следовательно, было бы также естественно предположить, что строки реализованы как ссылочные типы.

Но, пожалуй, хватит рассуждать о строках, давайте напишем какую-нибудь программу на JavaScript и поэкспериментируем. Если строки копируются и передаются по ссылке, мы сможем изменить содержимое строки по ссылке, хранящейся в другой переменной или переданной в функцию.

Однако, собравшись писать код для выполнения этого эксперимента, мы окажемся в тупике: нет способа изменить содержимое строки. Метод `charAt()` возвращает символ в данной позиции строки, но нет соответствующего метода `setCharAt()`. И это не недосмотр. Строки JavaScript намеренно сделаны *неизменными*, то есть в JavaScript нет синтаксиса, метода или свойства, позволяющего изменять символы в строке.

А раз строки нельзя изменять, то вопрос, с которого мы начали, остается нерешенным: нет способа понять, как передаются строки – по значению или по ссылке. Мы можем предположить, что, из соображений эффективности, JavaScript реализован таким образом, что строки передаются по ссылке, но в реальности это не имеет значения, так как это не оказывает никакого влияния на код.

11.2.3. Сравнение строк

И хотя мы не можем определить, как копируются и передаются строки – по значению или по ссылке, – мы можем написать код JavaScript, чтобы определить, как строки сравниваются. В примере 11.4 приведен код, позволяющий это сделать.

Пример 11.4. Сравниваются ли строки по значению или по ссылке?

```
// Определить, сравниваются ли строки по значению или по ссылке, просто.
// Мы сравниваем две совершенно разные строки, которые содержат одни и те же
// символы. Если они сравниваются по значению, то будут равны, но если они
// сравниваются по ссылке, - то нет:
var s1 = "hello";
var s2 = "hell" + "o";
if (s1 == s2) document.write("Строки сравниваются по значению");
```

Этот эксперимент демонстрирует, что строки сравниваются по значению. Некоторых программистов это может удивить. В C, C++ и Java строки являются ссылочными типами и сравниваются по ссылке. Для того чтобы сравнить фактическое содержимое двух строк, надо обратиться к специальному методу или функции. Однако JavaScript – это язык более высокого уровня, который понимает, что когда вы сравниваете строки, вам чаще всего требуется сравнить их по значению. Следовательно, несмотря на то что ради эффективности строки JavaScript (предположительно) копируются и передаются по ссылке, сравниваются они по значению.

11.2.4. Передача по значению и по ссылке: итог

В табл. 11.3 подводится итог обсуждению способов манипуляции с различными типами JavaScript.

Таблица 11.3. Манипуляции с типами данных JavaScript

Тип	Копируется	Передается	Сравнивается
Числа	По значению	По значению	По значению
Логическое	По значению	По значению	По значению
Строки	Неизменны	Неизменны	По значению
Объекты	По ссылке	По ссылке	По ссылке

11.3. Сборка мусора

Как объяснялось в главе 4, JavaScript освобождает память, занятую строками, объектами, массивами и функциями, которые больше не используются, при помощи сборки мусора. Это снимает с программиста ответственность за явное освобождение памяти, и это важная особенность, делающая программирование на JavaScript проще, чем, скажем, на C.

Ключевая особенность сборки мусора состоит в том, что сборщик мусора должен иметь возможность определить, когда память можно безопасно освободить. Очевидно, он ни при каких обстоятельствах не должен освобождать память, занятую значениями, которые еще используются, и должен освобождать память только от тех значений, которые больше не доступны, то есть от значений, к которым нельзя обратиться через какие-либо переменные, свойства объектов или элементы массивов в программе. Если вы любопытны, то можете заинтересоваться, как сборщик мусора отличает мусор, который должен быть собран, от значений, которые еще используются или могут использоваться потенциально. В следующих разделах раскрываются некоторые «анатомические» подробности.

11.3.1. Сборка мусора: «отметить и очистить»

В книгах по теории программирования о сборке мусора написано много, и тема эта не простая. Работа сборщика мусора на самом деле относится к деталям, специфическим для конкретной реализации, и в разных версиях языка реализована по-разному. Тем не менее почти все серьезные сборщики мусора основаны на какой-либо вариации базового алгоритма сборки мусора, известного как «отметить и очистить» (mark-and-sweep).

Сборщик мусора, работающий по этому алгоритму, периодически обходит список всех переменных среды JavaScript и отмечает любые значения, на которые ссылаются эти переменные. Если какие-либо из этих значений представляют собой объекты или массивы, он рекурсивно отмечает свойства объекта и элементы массива. Рекурсивно обходя это дерево или граф значений, сборщику мусора удается найти (и отметить) все отдельные значения, которые еще доступны. Затем он предполагает, что любые неотмеченные значения недоступны и, следовательно, представляют собой мусор.

Как только сборщик мусора завершит отметку всех доступных значений, он начинает фазу «очистки». На этом этапе он просматривает список всех переменных и освобождает те из них, которые не были отмечены. Классические сборщики мусора проходят полный цикл отметки и очистки одновременно, ощутимо замедляя систему. Более сложные вариации алгоритма осуществляют процесс относительно эффективно и выполняют сборку в фоновом режиме, не влияя на производительность системы.

Детали работы сборщика мусора зависят от реализации, и вам не требуется знать о нем что-либо для того, чтобы писать программы на JavaScript. Все современные реализации JavaScript основаны на каком-то из видов сборки мусора по алгоритму «отметить и очистить». Однако в JavaScript 1.1 в реализации Netscape 3 реализована несколько упрощенная схема, имеющая определенные недостатки. Тем, кто пишет код JavaScript, который должен быть совместим с Netscape 3, рекомендуется прочесть следующий раздел, описывающий недостатки сборщика мусора в этом броузере. В Netscape 2 применялась еще более простая технология сборки мусора с серьезными дефектами. Но этот броузер сейчас совершенно устарел, поэтому подробности здесь не рассматриваются.

11.3.2. Сборка мусора с помощью счетчика ссылок

В JavaScript 1.1, реализованном в Netscape 3, сборка мусора выполняется с помощью счетчика ссылок. Это значит, что любой объект (как пользовательский объект, созданный в коде JavaScript, так и встроенный объект HTML, созданный броузером) отслеживает число ссылок на него. Вспомните, что объекты присваиваются в JavaScript по ссылке, а не путем копирования их значений целиком.

Когда создается объект, и ссылка на него сохраняется в переменной, счетчик ссылок объекта становится равным 1. Когда ссылка на объект копируется и сохраняется в другой переменной, счетчик ссылок увеличивается до 2. Когда одна из двух переменных, содержащих эти ссылки, переписывается каким-либо новым значением, счетчик ссылок объекта снова уменьшается до 1. Если счетчик ссылок достигает нуля, ссылок на объект больше нет. Поскольку ссылку для копирования теперь взять негде, ссылок на объект в программе больше никогда не будет. Следовательно, JavaScript знает, что можно безопасно уничтожить объект и освободить связанную с ним память.

К сожалению, у счетчиков ссылок как у схемы сборки мусора есть недостатки. На самом деле некоторые даже не считают счетчик ссылок истинной сборкой мусора и оставляют этот термин для более совершенных алгоритмов, таких как «отметить и очистить». Счетчик ссылок – это простая форма сборки мусора, легко реализуемая и хорошо работающая во многих ситуациях. Однако в одном, очень важном случае счетчик ссылок не может корректно обнаружить и собрать весь мусор, и это необходимо иметь в виду.

Основной дефект счетчика ссылок имеет отношение к *циклическим ссылкам*. Если объект A содержит ссылку на объект B, а объект B – ссылку на объект A, существует цикл ссылок. Цикл также будет существовать, если, например, A ссылается на B, B ссылается на C, а C ссылается на A. В таких ситуациях внутри цикла всегда существует ссылка на любой его элемент. Поэтому даже если ни для одного из элементов цикла не осталось внешних ссылок, их счетчики ссылок никогда не получат значение меньше 1 и объекты никогда не будут удалены сборщиком мусора. Полный цикл может быть мусором, если нет возможности сослаться на какой-либо из этих объектов из программы, но так как они все ссылаются друг на друга, сборщик мусора на базе счетчика ссылок не может найти и освободить эту неиспользуемую память.

Неприятность с циклами – это цена, которую приходится платить за простоту схемы сборки мусора. Единственный способ обойти это препятствие связан с ручным вмешательством. Создавая цикл объектов, необходимо отдавать себе в этом отчет и принять меры, чтобы обеспечить очистку этих объектов при сборке мусора, когда они станут не нужны. Для того чтобы сборщик мусора увидел цикл объектов, следует разорвать цикл. Это можно сделать, выбрав один из объектов цикла и установив его свойство, ссылающееся на следующий объект, в `null`. Предположим, что A, B и C – это объекты, каждый из которых имеет свойство `next`, и значение этого свойства установлено таким образом, что объекты ссылаются друг на друга и образуют

цикл. Когда эти объекты более не используются, можно разорвать цикл, присвоив `A.next` значение `null`. Это значит, что на объект B больше нет ссылки из A, поэтому его счетчик ссылок может уменьшиться до нуля и сборщик мусора сможет освободить объект B. Как только он будет освобожден, он более не будет ссылаться на C, счетчик ссылок C сможет уменьшиться до нуля и объект C может быть освобожден. Так как C освобожден, может быть освобожден и объект A.

Заметьте, что это, конечно, не может произойти, если A, B и C хранятся в глобальных переменных в окне, которое еще открыто, так как переменные A, B и C все еще ссылаются на объекты. Если бы они были локальными переменными в функции и их цикл был бы разорван до завершения функции, то они могли бы быть освобождены. Но если объекты хранятся в глобальных переменных, ссылки на них остаются до тех пор, пока не будет закрыто содержащее их окно. В этом случае, для того чтобы их освободить принудительно, надо разорвать цикл и установить все переменные в `null`:

```
A.next = null;     // Разрываем цикл
A = B = C = null;  // Удаляем последние оставшиеся внешние ссылки
```

11.4. Лексический контекст и вложенные функции

Функции в JavaScript относятся к лексическому контексту, а не к динамической области видимости. Это значит, что они работают в том контексте, в котором определены, а не в том, откуда они вызваны. До JavaScript 1.2 функции могли определяться только в глобальной области видимости, и лексический контекст не создавал проблем: все функции исполнялись в одном и том же контексте (в цепочке областей видимости также присутствовал объект вызова функции).

Однако в JavaScript 1.2 и более поздних версиях функции могут определяться везде, и возникают сложные вопросы, связанные с областями видимости. Рассмотрим, например, функцию `g`, определенную внутри функции `f`. Функция `g` всегда исполняется в контексте функции `f`. Цепочка областей видимости функции `g` включает три объекта: ее собственный объект вызова, объект вызова функции `f` и глобальный объект. Поведение вложенных функций вполне понятно, когда они вызываются в том же контексте, в котором определены. Например, следующий код не делает ничего особенно удивительного:

```
var x = "global";
function f() {
    var x = "local";
    function g() { alert(x); }
    g();
}
f();  // Вызов этой функции выдает "local"
```

Однако в JavaScript функции – это данные, такие же как и другие типы значений, поэтому они могут возвращаться из функций, присваиваться свойствам объекта, храниться в массивах и т. д. Это не приводит ни к чему удивительному, за исключением тех случаев, когда функция возвращает вложенную функцию. Рассмотрим следующий код, который включает функцию, возвращающую вложенную функцию. При каждом своем вызове она возвращает функцию. Код возвращаемой функции всегда один и тот же, но контекст, в котором она создается, при каждом вызове слегка различается, так как различаются значения аргументов, переданных внешней функции. Если мы сохраним возвращаемые функции в массиве и вызовем каждую из них, то увидим, что они возвращают разные значения. Все функции содержат один и тот же код и вызываются из одного и того же контекста, поэтому единственный фактор, который может приводить к тому, что возвращаются различные значения, – это контекст, в котором функции были определены.

```
// Эта функция возвращает функцию при каждом своем вызове.
// Контекст определения функции отличается  для каждого вызова
function makefunc(x) {
    return function() { return x; }
}

// Вызывает makefunc() несколько раз, и сохраняет результаты в массиве:
var a = [makefunc(0), makefunc(1), makefunc(2)];

// Теперь вызываем эти функции и отображаем их значения
// Хотя тела всех функций одинаковы, контекст отличается
// и при каждом вызове возвращаются различные значения.
alert(a[0]());  // Отображает 0
alert(a[1]());  // Отображает 1
alert(a[2]());  // Отображает 2
```

Результат работы этого кода может показаться удивительным. Однако это ожидаемые результаты строгого применения правила лексического контекста: функция выполняется в том контексте, в котором она определена. Этот контекст включает состояние локальных переменных и аргументов. Даже несмотря на то что значения локальных переменных и аргументов функции временны, их состояние замораживается и становится частью лексического контекста любой функции, определенной в то время, когда они были актуальны. Чтобы лексический контекст работал с вложенными функциями, реализация JavaScript должна использовать *замыкание* (*closure*), которое можно рассматривать как комбинацию определения функции и цепочки областей видимости, актуальных в момент определения функции.

11.5. Конструктор Function() и функциональные литералы

Как мы видели в главе 7, имеется два способа определения функций, отличных от обычной инструкции `function`. В JavaScript 1.1 функции могут определяться с помощью конструктора `Function()`, а в JavaScript 1.2 и более поздних версиях – с помощью функциональных литералов. Необходимо помнить о некоторых важных различиях между этими двумя технологиями.

Во-первых, конструктор `Function()` позволяет динамически создавать и компилировать код JavaScript во время выполнения. В отличие от него функциональные литералы – это статическая часть структуры программы, такая же как инструкции `function`.

Во-вторых, что следует из первого различия, конструктор `Function()` анализирует тело функции и создает новый функциональный объект при каждом его вызове. Если вызов конструктора помещен внутрь цикла или часто вызываемой функции, то данный процесс может быть неэффективным. Функциональный литерал или вложенная функция не перекомпилируется каждый раз. Новый функциональный объект также не создается каждый раз, когда встречается функциональный литерал. (Тем не менее, как было отмечено ранее, может потребоваться новое замыкание для отслеживания различий в лексическом контексте, в котором определена функция.)

Третье различие между конструктором `Function()` и функциональными литералами состоит в том, что функции, созданные с помощью конструктора `Function()`, не учитывают лексический контекст; вместо этого они всегда компилируются как функции верхнего уровня, что показано в следующем коде:

```
var y = "глобальная";
function constructFunction() {
    var y = "локальная";
    return new Function("return y");  // Не использует локальную область видимости!
}
// Эта строка выводит слово "глобальная", т.к. функция, возвращаемая конструктором
// Function(), не использует локальную область видимости. Если бы вместо этого
// использовался функциональный литерал, строка выдала бы слово "локальная".
alert(constructFunction()());  // Отображает "глобальная"
```

11.6. Несовместимость Netscape JavaScript 1.2

JavaScript 1.2 для Netscape был реализован как часть броузера Netscape 4.0, когда спецификация ECMAScript v1 еще не была завершена. Разработчики из Netscape лишь догадывались о том, что должно быть в спецификации, и на основе этих догадок изменили поведение интерпретатора. Поскольку эти изменения были несовместимы с предыдущими версиями JavaScript, они были реализованы только для случаев, когда явно запрашивалось применение JavaScript 1.2. (В веб-броузерах это достигается установкой атрибута `language` тега HTML `<script>` равным «JavaScript 1.2».). Это был отличный

способ ввести новые возможности, не нарушая работу старых сценариев. К сожалению, когда работа над ECMAScript v1 завершилась, новое поведение, предполагавшееся разработчиками JavaScript 1.2, оказалось несовместимым с JavaScript 1.1 и несоответствующим спецификации ECMAScript.

Для совместимости со сценариями, полагающимися на нестандартное поведение JavaScript 1.2, все следующие реализации JavaScript от Netscape сохранили эти особенности при явном указании версии 1.2. Но если потребовать версию, более высокую, чем 1.2 (например, указав `language="JavaScript1.3"`), то будет получено поведение, совместимое с ECMAScript. Поскольку это особое поведение присутствует только в реализациях JavaScript от Netscape, не следует основывать на них свои сценарии и лучше всего никогда явно не указывать версию 1.2. Тем не менее иногда применение JavaScript 1.2 необходимо, и ниже перечислены особенности этой версии:

- Операторы равенства и неравенства ведут себя так же, как операторы идентичности и неидентичности. Другими словами, `==` работает так же, как `===`, а `!=` работает, как `!==`.
- Стандартный метод `Object.toString()` отображает значения всех свойств, определенных в объекте, возвращая строку, отформатированную в соответствии с синтаксисом объектных литералов.
- Метод `Array.toString()` разделяет элементы массива запятыми и пробелами, вместо использования только запятых, и возвращает список элементов, заключенный в квадратные скобки. Кроме того, строковые элементы массива заключаются в кавычки, так что результатом является строка в виде допустимого литерала массива.
- Когда конструктору `Array()` передается единственный числовой аргумент *n*, он возвращает массив с единственным элементом *n*, а не массив длиной *n*.
- Значением объекта-массива в числовом контексте является его длина. В логическом контексте он принимает значение `false`, если длина равна 0, и `true` в противном случае.
- Метод `Array.push()` возвращает последнее добавленное в массив значение, а не новую длину массива.
- Когда метод `Array.splice()` вырезает единственный элемент *x*, он возвращает сам элемент *x*, а не массив, содержащий *x* в качестве его единственного элемента. Когда `splice()` не удаляет ни одного элемента из массива, он не возвращает ничего, вместо возвращения пустого массива.
- Когда метод `String.substring()` вызывается с начальной позицией, большей, чем указанная конечная позиция, он возвращает пустую строку, вместо корректной перестановки аргументов и возвращения подстроки между ними.
- Особое поведение метода `String.split()` унаследовано из Perl: если указанный символ-разделитель – это одиночный пробел, метод отбрасывает все ведущие и замыкающие пробельные символы строки, а затем выполняет разбиение оставшейся ее части.

Часть II. Клиентский JavaScript

В данной части книги в главах с 12 по 22 приведено описание JavaScript в том виде, в котором он реализован в веб-броузерах. Эти главы вводят много новых объектов JavaScript, представляющих веб-броузер и содержимое HTML-документов. Тщательное изучение многочисленных примеров, иллюстрирующих типичные применения описываемых клиентских объектов, будет для вас полезным.

- Глава 12 «JavaScript в веб-броузерах»
- Глава 13 «Окна и фреймы»
- Глава 14 «Объект Document»
- Глава 15 «Формы и элементы форм»
- Глава 16 «Сценарии и cookies»
- Глава 17 «Объектная модель документа »
- Глава 18 «Каскадные таблицы стилей и Dynamic HTML»
- Глава 19 «События и обработка событий»
- Глава 20 «Приемы обеспечения совместимости»
- Глава 21 «Безопасность в JavaScript»
- Глава 22 «Совместное применение Java и JavaScript»

12

JavaScript в веб-броузерах

Часть I этой книги была посвящена базовому языку JavaScript. Теперь мы перейдем к тому JavaScript, который используется в веб-броузерах, и обычно называется «клиентский JavaScript» (client-side JavaScript).[1] Большинство примеров, которые мы видели до этого, будучи корректным кодом JavaScript, не имели определенного контекста; они были фрагментами JavaScript, не предназначенными для запуска в какой-либо определенной среде. Эта глава предоставляет такой контекст. Она начинается с абстрактного введения в среду программирования веб-броузера и в базовые концепции клиентского JavaScript. Затем в ней обсуждается, каким образом код JavaScript фактически встраивается в HTML-документы, чтобы он мог быть запущен веб-броузером. И в конце этой главы подробно рассказывается о том, как программы JavaScript исполняются в веб-броузере.

12.1. Среда веб-броузера

Чтобы понять, что такое клиентский JavaScript, необходимо понять концептуальную основу среды программирования, предоставляемой веб-броузером. Следующие разделы вводят три основные компоненты этой среды программирования:

---

[1] Термин «client-side JavaScript» остался с тех времен, когда JavaScript применялся только в двух местах: веб-броузерах (клиентах) и веб-серверах. Поскольку JavaScript в качестве языка сценариев распространяется во все большем количестве сред, термин «client-side» имеет все меньше и меньше смысла, так как не определяет *принадлежность* клиентской стороны. Тем не менее в этой книге мы будем по-прежнему употреблять этот термин.

- Объект Window, который служит в качестве глобального объекта и глобального контекста исполнения для кода клиентского JavaScript.
- Иерархия объектов клиентского JavaScript и объектная модель документа, которая образует ее часть.
- Управляемая событиями модель программирования.

12.1.1. Окно как глобальный контекст исполнения

Основная задача веб-броузера состоит в отображении HTML-документа в окне. В клиентском JavaScript объект Document представляет HTML-документ, а объект Window представляет окно (или фрейм), отображающее этот документ. Хотя в клиентском JavaScript оба этих объекта важны, объект Window более важен по одной существенной причине: это глобальный объект в клиентском программировании.

Вспомните из главы 4, что в любой реализации JavaScript на вершине цепочки областей видимости всегда расположен глобальный объект; свойства глобального объекта являются глобальными переменными. В клиентском JavaScript объект Window – это глобальный объект. Объект Window определяет несколько свойств и методов, позволяющих нам манипулировать окном веб-броузера. Он также определяет свойства, ссылающиеся на другие важные объекты, такие как свойство `document` для объекта Document. И наконец, объект Window имеет два свойства для ссылки на себя – `window` и `self`. Любая из этих глобальных переменных может использоваться для ссылки непосредственно на объект Window.

Поскольку объект Window – это глобальный объект клиентского JavaScript, все глобальные переменные определяются как свойства окна. Например, следующие две строки кода выполняют по существу одно и то же действие:

```
var answer = 42;      // Объявляем и инициализируем глобальную переменную
window.answer = 42;   // Создаем новое свойство объекта Window
```

Объект Window представляет окно веб-броузера или фрейм внутри окна. Для клиентского JavaScript окна верхнего уровня и фреймы по существу эквивалентны. Часто создаются приложения JavaScript, работающие с несколькими фреймами, и возможны, хотя и не так распространены, приложения, работающие с несколькими окнами. Каждое окно или фрейм приложения имеет уникальный объект Window и определяет уникальный контекст исполнения для кода клиентского JavaScript. Другими словами, глобальная переменная, объявленная кодом JavaScript в одном фрейме, не является глобальной в другом фрейме. Однако второй фрейм *может* обращаться к глобальной переменной первого фрейма. Мы увидим, как это делается, когда будем рассматривать данную тему более подробно в главе 13.

12.1.2. Иерархия объектов клиентского JavaScript и объектная модель документа

Мы видели, что объект Window – это ключевой объект в клиентском JavaScript. Все остальные объекты присоединены к нему. Например, любой объект Window содержит свойство `document`, ссылающееся на связанный с окном объект Document, и свойство `location`, ссылающееся на связанный с окном объект Location. Объект Window также содержит массив `frames[]`, ссылающийся на объекты Window, представляющие фреймы исходного окна. Следовательно, `document` представляет объект Document текущего окна, а `frames[1].document` ссылается на объект Document второго дочернего фрейма текущего окна.

Объект, на который ссылается текущее окно или какой-либо другой объект Window, может сам ссылаться на другие объекты. Например, в каждом объекте Document имеется массив `forms[]`, содержащий объекты Form, которые представляют любые присутствующие в документе HTML-формы. Для ссылки на одну из этих форм можно использовать выражение

```
window.document.forms[0]
```

Продолжим тот же пример: в каждом объекте Form имеется массив `elements[]`, содержащий объекты, которые представляют различные HTML-элементы (поля ввода, кнопки и т. д.), присутствующие внутри формы. В некоторых случаях программисту приходится писать код, ссылающийся на объект в конце всей цепочки объектов, получая примерно такие сложные выражения:

```
parent.frames[0].document.forms[0].elements[3].options[2].text
```

Мы увидели, что объект Window – это глобальный объект в начале цепочки областей действия и что все клиентские объекты в JavaScript доступны как свойства других объектов. Это значит, что имеется иерархия объектов JavaScript, в корне которой находится объект Window. Эта иерархия показана на рис. 12.1. Внимательно изучите схему – понимание этой иерархии и составляющих ее объектов является ключом для успешного программирования в клиентском JavaScript. Большинство оставшихся глав этой книги посвящены конкретизации описаний объектов, изображенных на данном рисунке.

Обратите внимание, что на рис. 12.1 показаны только свойства объектов, ссылающиеся на другие объекты. Большинство объектов, изображенных на диаграмме, имеют немало свойств, которые здесь не показаны.

Многие объекты, изображенные на этом рисунке, происходят от объекта Document. Это поддерево большой иерархии клиентского JavaScript известно как *объектная модель документа* (*Document Object Model, DOM*) и интересно тем, что на нем сконцентрировались усилия по стандартизации. Рис. 12.1 иллюстрирует объекты документа, которые стали стандартом «де-факто», так как они согласованно реализованы всеми основными броузерами. Все

вместе они известны как DOM Level 0, так как образуют базовый уровень функциональности документа, на которую программисты JavaScript могут полагаться во всех броузерах. Эти основные объекты документа рассмотрены в главах 14 и 15. Более продвинутая объектная модель документа, стандартизованная W3C, является темой глав 17 и 18.

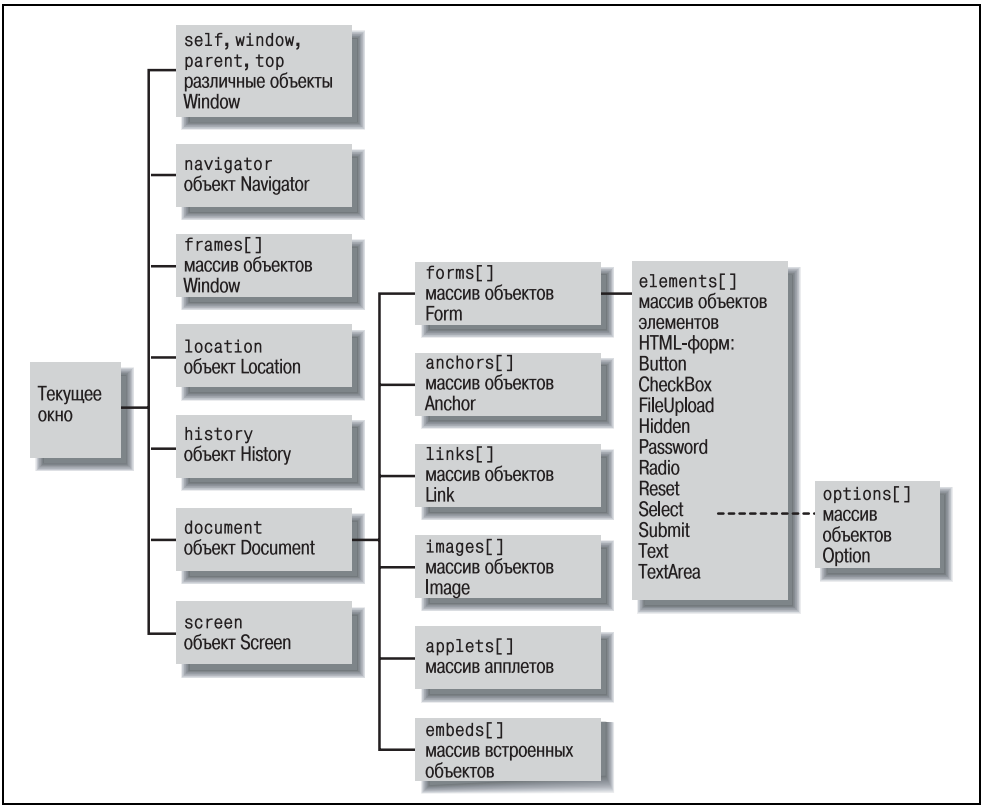

Рис. 12.1. *Иерархия объектов клиентского JavaScript и DOM Level 0*

12.1.3. Управляемая событиями модель программирования

В прошлом компьютерные программы часто работали в пакетном режиме – читали пакет данных, выполняли какие-то вычисления, а затем выводили результат. Позднее, с введением разделения времени и текстовых терминалов, стали возможными ограниченные виды интерактивности – программа могла запросить от пользователя данные, а тот мог их ввести. Компьютер затем обрабатывал данные и выводил на экран результат.

С появлением графических дисплеев и указывающих устройств, таких как мыши, ситуация изменилась. Программы в основном стали управляться со-

бытиями; они реагируют на асинхронный пользовательский ввод в виде щелчков мыши и нажатий клавиш, способ интерпретации которых зависит от положения указателя мыши. Веб-броузер – как раз такая графическая среда. HTML-документ содержит в себе графический пользовательский интерфейс (GUI), поэтому клиентский JavaScript использует управляемую событиями модель программирования.

Вполне возможно написать статическую программу JavaScript, не принимающую пользовательских данных и делающую всегда одно и то же. Иногда такие программы полезны. Однако чаще мы пишем динамические программы, взаимодействующие с пользователем. Чтобы это сделать, у нас должна быть возможность реагировать на его действия.

В клиентском JavaScript веб-броузер уведомляет программы о действиях пользователя, генерируя *события*. Имеются различные типы событий, такие как нажатие клавиш, перемещение мыши и т. д. Когда происходит событие, веб-броузер пытается вызвать соответствующую функцию-*обработчик события* для реакции на него. Поэтому для написания динамических, интерактивных программ клиентского JavaScript мы должны определить нужные обработчики событий и зарегистрировать их в системе, чтобы броузер мог вызывать их в нужные моменты.

Тем, кто еще не знаком с управляемой событиями моделью программирования, придется потратить немного времени, чтобы к ней привыкнуть. В старой модели программист писал единый, монолитный блок кода, выполнение которого осуществлялось в каком-либо определенном порядке, и который работал от начала до конца. Управляемое событиями программирование переворачивает эту модель с ног на голову. В управляемом событиями программировании создается несколько независимых (но взаимодействующих между собой) обработчиков событий. Программист не вызывает их непосредственно, а позволяет системе вызывать их в нужный момент. Так как обработчики запускаются от действий пользователя, они могут исполняться в непредсказуемые, асинхронные моменты времени. Большую часть времени программа вообще не работает, а просто ожидает, пока система вызовет один из ее обработчиков событий.

В следующем разделе объясняется, как встроить код JavaScript в HTML-файл, как определять и статические блоки кода, работающие синхронно от начала до конца, и обработчики событий, вызываемые системой асинхронно. События и их обработку мы обсудим подробнее в главе 19.

12.2. Встраивание JavaScript в HTML

Код клиентского JavaScript встраивается в HTML-документы несколькими способами:

- Между парой тегов `<script>` и `</script>`
- Из внешнего файла, заданного атрибутом `src` тега `<script>`

- В обработчик события, заданный как значение HTML-атрибута, такого как `onclick` или `onmouseover`
- Как тело URL, использующего специальный протокол `javascript:`

В следующих разделах каждая из этих технологий встраивания JavaScript документирована более подробно. В них описываются все способы включения JavaScript в веб-страницы, то есть объясняется допустимая структура программ JavaScript на стороне клиента.

12.2.1. Тег <script>

Сценарии клиентского JavaScript представляют собой часть HTML-файла и находятся между тегами `<script>` и `</script>`. Количество инструкций JavaScript между этими тегами может быть любым; инструкции выполняются по порядку, как часть процесса загрузки документа. Теги `<script>` могут присутствовать как в разделе `<head>`, так и в разделе `<body>` HTML-документа.

Один HTML-документ может содержать любое количество неперекрывающихся пар тегов `<script>` и `</script>`. Эти несколько отдельных сценариев исполняются в том порядке, в котором они присутствуют в документе. Хотя отдельные сценарии в одном файле исполняются в различные моменты в процессе загрузки и анализа HTML-файла, они представляют собой части одной программы на JavaScript: функции и переменные, определенные в одном сценарии, доступны всем сценариям, находящимся в том же файле. Например, в HTML-странице может быть следующий сценарий:

```
<script>var x = 1;</script>
```

Далее на той же HTML-странице вы можете ссылаться на `x`, даже в другом блоке сценария. Контекстом является HTML-страница, а не блок сценария:

```
<script>document.write(x);</script>
```

`document.write()` — это важный и часто применяемый метод. В данном примере он вставляет результат своей работы в документ в месте расположения сценария. Когда сценарий завершает исполнение, анализатор HTML возобновляет работу и начинает анализировать текст, произведенный методом `document.write()`.

В примере 12.1 показан HTML-файл, включающий простую программу на JavaScript. Обратите внимание на различие между этим примером и многими фрагментами кода, показанными в этой книге ранее: пример интегрирован в HTML-файл и имеется понятный контекст, в котором он работает. Обратите также внимание на атрибут `language` в теге `<script>`. Его описание приведено в следующем разделе.

Пример 12.1. Простая программа JavaScript в HTML-файле

```
<html>
<head>
<title>Сегодняшняя дата</title>
```

```
    <script language="JavaScript">
    // Определяем функцию для дальнейшего использования
    function print_todays_date() {
        var d = new Date();                  // Получаем текущую дату и время
        document.write(d.toLocaleString());  // Вставляем это в документ
    }
    </script>
</head>
<body>
Дата и время:<br>
<script language="JavaScript">
  // Теперь вызываем функцию, определенную выше
  print_todays_date();
</script>
</body>
</html>
```

12.2.1.1. Атрибуты language и type

Хотя JavaScript, пожалуй, применяется чаще других языков клиентских сценариев, он не единственный. Необязательный атрибут `language` тега `<script>` сообщает веб-броузеру, на каком языке написан сценарий,. Броузеры, понимающие указанный язык, выполняют сценарий; броузеры, не понимающие этот язык, сценарий пропускают.

Если код написан на JavaScript, то атрибут `language` выглядит так:

```
<script language="JavaScript">
    // Здесь располагается код JavaScript
</script>
```

А если, например, сценарий написан на языке Microsoft Visual Basic Scripting Edition[1], то атрибут должен быть таким:

```
<script language="VBScript">
  ' Здесь располагается код VBScript (' это символ комментария, как // в JavaScript)
</script>
```

JavaScript – это стандартный язык сценариев для Всемирной паутины, и если атрибут `language` пропущен, то и Netscape и Internet Explorer предполагают, что ваши сценарии написаны на JavaScript.

Спецификация HTML 4 стандартизует тег `<script>`, но отвергает атрибут `language`, так как стандартный набор имен языков сценариев не определен. Спецификация отдает предпочтение атрибуту `type`, задающему язык сценариев

[1] Известен еще как VBScript. Единственный броузер, поддерживающий VBScript, – это Internet Explorer, поэтому написанные на этом языке сценарии не являются переносимыми. Взаимодействие VBScript с объектами HTML осуществляется так же, как в JavaScript, однако сам базовый язык имеет отличный от JavaScript синтаксис. В этой книге VBScript не описан.

как MIME-тип. Поэтому в теории предпочтительный способ внедрения кода JavaScript – это тег, который выглядит следующим образом:

```
<script type="text/javascript">
```

На практике атрибут language все еще поддерживается лучше, чем новый атрибут type.

В спецификации HTML 4 также определен стандартный (и полезный) способ задания языка программирования по умолчанию для всех сценариев данного HTML-файла. Если JavaScript будет единственным языком в файле, просто включите следующую строку в тег <head> документа:

```
<meta http-equiv="Content-Script-Type" content="text/javascript">
```

После этого можно спокойно использовать сценарии JavaScript без указания атрибутов language или type.

Поскольку JavaScript – это язык сценариев по умолчанию, тем, кто программирует на нем, никогда не требуется задавать язык, на котором написан сценарий, при помощи атрибута language. Однако у этого атрибута есть важное вторичное назначение: он может служить для указания, версии JavaScript, требуемой для интерпретации сценария. Если атрибут language="JavaScript" задан, то сценарий будет выполнен любым броузером, поддерживающим JavaScript. Но предположим, что в сценарий включены средства обработки исключений JavaScript 1.5. Чтобы избежать синтаксических ошибок в броузерах, не поддерживающих эту версию языка, можно встроить свой сценарий с помощью следующего тега:

```
<script language="JavaScript1.5">
```

При наличии такого тега сценарий будет выполняться только броузерами, поддерживающими JavaScript 1.5 (и его средства обработки исключений); все остальные будут его игнорировать.

Отдельного упоминания заслуживает значение «JavaScript1.2» атрибута language. Когда Netscape 4 готовился к выпуску, оказалось, что выходящий стандарт ECMA-262 потребует внесения некоторых несовместимых изменений в определенные средства языка. Чтобы эти несовместимые изменения не нарушали работу существующих сценариев, разработчики JavaScript в Netscape приняли разумную меру предосторожности, определив, что эти изменения будут действовать, только если в атрибуте language будет явно указано «JavaScript1.2». К сожалению, стандарт ECMA не был завершен до выхода Netscape 4, а после выхода финальной версии предложенные несовместимые изменения языка были убраны из стандарта. Поэтому указание language="JavaScript1.2" заставляет Netscape 4 вести себя несовместимым с предыдущими броузерами и спецификацией ECMA образом. (Полное описание этой несовместимости приведено в разделе 11.6 «Несовместимость Netscape JavaScript 1.2».) По этой причине желательно избегать указания «JavaScript1.2» в качестве значения атрибута language.

12.2.1.2. Тег </script>

В какой-то момент вам может потребоваться, чтобы метод document.write() выводил сценарий в какое-либо другое окно или фрейм броузера. А тогда для завершения генерируемого сценария потребуется вывести тег </script>. Здесь необходима осторожность – HTML-анализатор не пытается понять код JavaScript и, встретив строку «/script» даже внутри кавычек, он предполагает, что это закрывающий тег выполняемого в данный момент сценария. Для того чтобы обойти это препятствие, разбейте тег на фрагменты и запишите его в виде выражения, например "</" + "script>":

```
<script>
f1.document.write("<script>");
f1.document.write("document.write('<h2>Это сценарий, заключенный в кавычки</h2>')");
f1.document.write("</" + "script>");
</script>
```

В качестве альтернативы можно указать обратную косую черту перед прямой косой чертой в теге:

```
f1.document.write("<\/script>");
```

12.2.1.3. Атрибут defer

Стандарт HTML 4 определяет атрибут тега <script>, который пока употребляется не часто, но может быть полезным. Как кратко говорилось выше, сценарий может вызывать метод document.write() для динамического добавления содержимого в документ. Поэтому когда HTML-анализатор встречает сценарий, он должен прекратить анализ и ожидать его завершения.

Если сценарий не выполняет какого-либо вывода в документ, например, он, определяет функцию, но нигде не вызывает document.write(), то с помощью атрибута defer тега <script> можно сообщить броузеру, чтобы он спокойно продолжал обработку HTML-документа и отложил исполнение сценария до тех пор, пока не будет найден сценарий, выполнение которого не может быть отложено. Это может привести к улучшению производительности в броузерах, способных использовать преимущества атрибута defer. Обратите внимание, что у атрибута defer не может быть значения; он просто должен присутствовать в теге:

```
<script defer>
    // Любой код JavaScript, не вызывающий document.write()
</script>
```

12.2.2. Включение файлов JavaScript

В JavaScript 1.1 тег <script> поддерживает атрибут src. Значение этого атрибута задает URL файла, содержащего код JavaScript. Он применяется следующим образом:

```
<script src="../../javascript/util.js"></script>
```

Файл JavaScript обычно имеет расширение *.js* и содержит код JavaScript в чистом виде, без тегов `<script>` или любого другого HTML-кода.

Тег `<script>` с атрибутом `src` ведет себя точно так же, как если бы содержимое указанного файла JavaScript находилось непосредственно между тегами `<script>` и `</script>`. Любой код, указанный между этими тегами, будет игнорироваться броузерами, поддерживающими атрибут `src` (но, однако, исполняется броузерами, не понимающими этот атрибут, например Netscape 2,). Обратите внимание, что закрывающий тег `</script>` обязателен даже в том случае, когда указан атрибут `src` и между тегами отсутствует код JavaScript.

Использование тега с атрибутом `src` дает ряд преимуществ:

- Упрощаются HTML-файлы, так как из них можно убрать большие блоки кода JavaScript.
- Функцию или другой код JavaScript, используемый несколькими различными HTML-файлами, можно держать в одном файле и считывать при необходимости. Это уменьшает объем занимаемой дисковой памяти и намного облегчает поддержку кода.
- Когда функции JavaScript используются несколькими страницами, помещение кода в отдельный файл позволяет броузеру кэшировать его и тем самым ускорять загрузку. Когда код JavaScript совместно используется несколькими страницами, экономия времени, достигаемая за счет кэширования, явно перевешивает небольшую задержку, требуемую броузеру для открытия отдельного сетевого соединения для загрузки файла JavaScript при первом запросе на загрузку этого кода.
- Атрибут `src` принимает в качестве значения произвольный URL, поэтому программа на JavaScript или веб-страница с одного веб-сервера может воспользоваться кодом (например, из библиотеки подпрограмм), предоставляемым другими веб-серверами.

12.2.3. Обработчики событий

Код JavaScript, расположенный в теге `<script>`, исполняется один раз, когда содержащий его HTML-файл читается в веб-броузер. Такие статические сценарии не могут динамически реагировать на действия пользователя. Более динамические программы определяют обработчики событий, автоматически вызываемые веб-броузером при возникновении определенных событий. Например, при нажатии кнопки формы. События в клиентском JavaScript генерируются объектами HTML (такими как кнопки), поэтому обработчики событий определяются как атрибуты этих объектов. Например, чтобы определить обработчик события, который вызывается, когда пользователь щелкает по флажку в форме, код обработчика указывается в качестве атрибута HTML-тега, определяющего флажок:

```
<input type="checkbox" name="opts" value="ignore-case"
       onclick="ignore-case = this.checked;"
>
```

Здесь нас интересует атрибут onclick.[1] Строковое значение атрибута onclick может содержать одну или несколько инструкций JavaScript. Если имеется несколько инструкций, они должны отделяться друг от друга точками с запятой. Когда с флажком происходит указанное событие – в данном случае щелчок кнопкой мыши, – исполняется код JavaScript, указанный в этой строке.

В определение обработчика события можно включать любое количество инструкций JavaScript, но обычно обработчик определяют как функцию, заключенную между тегами <script> и </script>, если он требует больше пары простых инструкций. Тогда можно вызвать данную функцию из обработчика события. Это позволяет держать большую часть кода JavaScript в тегах <script> и уменьшает необходимость в смешивании JavaScript и HTML.

Намного более подробно события и их обработчики обсуждаются в главе 19, но их применение рассматривается и раньше во многих примерах. Там содержится полный список обработчиков событий, но наиболее распространенные из них мы перечислим здесь:

onclick

Этот обработчик поддерживается всеми подобными кнопкам элементами форм, а также тегами <a> и <area>. Он вызывается, когда пользователь щелкает по этому элементу. Если обработчик onclick возвращает false, броузер не выполняет стандартное действие, связанное с кнопкой или ссылкой; например, он не открывает ссылку (для тега <a>) или не передает данные формы (для кнопки Submit).

onmousedown, onmouseup

Эти два обработчика во многом похожи на onclick, но вызываются по отдельности, когда пользователь нажимает и отпускает кнопку мыши. Элементы документа, поддерживающие onclick, также поддерживают и эти обработчики. В IE 4 и Netscape 6 эти обработчики фактически поддерживаются почти всеми элементами документа.

onmouseover, onmouseout

Эти два обработчика события вызываются, когда указатель мыши, соответственно, проходит над элементом документа или выходит за его пределы. Чаще всего они применяются с тегами <a>. Если обработчик onmouseover тега <a> возвращает true, броузер не отображает URL ссылки в строке состояния.

onchange

Этот обработчик события поддерживается элементами <input>, <select> и <textarea>. Он вызывается, когда пользователь изменяет значение, отображаемое элементом, а затем перемещает фокус с помощью клавиши табуляции либо другим способом.

---

[1] Имена всех HTML-атрибутов обработчиков событий начинаются с «on».

onsubmit, onreset

Эти обработчики событий поддерживаются тегом <form> и вызываются, когда форма должна быть подтверждена или очищена. Они могут возвращать false для отмены действия кнопки. Обработчик onsubmit обычно служит для проверки данных формы на стороне клиента.

Реализацию обработчиков событий можно найти в интерактивном сценарии выплат по закладной в примере 1.3. HTML-форма в этом примере содержит несколько атрибутов обработчиков событий. Тело этих обработчиков просто: они лишь вызываютт функцию calculate(), определенную в другом месте внутри тега <script>.

12.2.4. JavaScript в URL

Другой способ исполнения кода JavaScript на стороне клиента – это URL, следующий за спецификатором псевдопротокола javascript:. Этот специальный тип протокола обозначает, что тело URL представляет собою произвольный код JavaScript, который должен быть выполнен интерпретатором JavaScript. Если код JavaScript в URL javascript: содержит несколько инструкций, они должны быть отделены друг от друга точками с запятой. Подобный URL может выглядеть, например, так:

```
javascript:var now = new Date(); "<h1>Время:</h1>" + now;
```

Когда броузер загружает такой URL, он исполняет содержащийся в URL код и использует строковое значение последней инструкции JavaScript в качестве содержимого нового отображаемого документа. Это строковое значение может содержать HTML-теги и форматируется и отображается точно так же, как любой другой документ, загруженный в броузер.

URL JavaScript может также содержать инструкции JavaScript, выполняющие действия, но не возвращающие значения. Например:

```
javascript:alert("Hello World!")
```

Когда загружается подобный URL, броузер исполняет код JavaScript, но так как значения для вывода в новом документе нет, он не изменяет текущий документ.

Часто возникает необходимость использовать URL javascript: для исполнения некоторого кода без изменения текущего отображаемого документа. При этом необходимо, чтобы последняя инструкция в URL не возвращала значение. Один из способов обеспечить это состоит в том, чтобы посредством оператора void явно указать неопределенное возвращаемое значение. Просто поместите инструкцию void 0 в конец вашего URL javascript:. Вот, например, URL, открывающий новое пустое окно броузера без изменения содержимого текущего окна:

```
javascript:window.open("about:blank"); void 0;
```

Без оператора void в этом URL значение, возвращаемое методом Window.open(), должно быть преобразовано в строку и показано, и текущий документ будет заменен документом, в котором присутствует что-то вроде

```
[object Window]
```

URL javascript: можно применять везде, где используется обычный URL. Один из важных способов применения этого синтаксиса – его ввод непосредственно в поле Location броузера, где можно проверять произвольный код JavaScript без необходимости открывать редактор и создавать HTML-файл с этим кодом.

URL javascript: могут помещаться в закладки, где они образуют минипрограммы JavaScript, или «букмарклеты», которые могут легко запускаться из меню или из панели закладок.

URL javascript: могут также выступать в качестве значения атрибута href гиперссылки. Когда пользователь щелкает по такой ссылке, исполняется указанный код JavaScript. А если указать URL javascript: в качестве значения атрибута action тега <form>, то код JavaScript исполняется при подтверждении пользователем формы. В этих контекстах URL javascript: по существу заменяет собой обработчик события.

Есть несколько ситуаций, когда URL javascript: может применяться с объектами, не поддерживающими обработчики событий. Например, тег <area> не поддерживает на платформах Windows в Netscape 3 событие onclick (но поддерживает в Netscape 4). Поэтому для того чтобы выполнить код JavaScript при щелчке пользователем по клиентской карте изображения в Netscape 3, следует использовать URL javascript:.

12.2.5. JavaScript в нестандартных контекстах

И Netscape и Microsoft реализовали в своих броузерах собственные фирменные расширения, и иногда код JavaScript встречается в контекстах, отличных от описанных здесь. Например, Internet Explorer позволяет определять обработчики событий в теге <script>, использующем специальные атрибуты for и event. Netscape 4 позволяет применять JavaScript в качестве альтернативного синтаксиса определения таблиц стилей CSS в теге <style>. Netscape 4 также расширяет синтаксис сущностей HTML и позволяет использовать JavaScript внутри сущностей (но только внутри значений HTML-атрибутов). Это может приводить к появлению следующего кода HTML:

```
<table border="&{getBorderWidth()};">
```

Кроме того, Netscape 4 также поддерживает форму условных комментариев на основе синтаксиса сущностей JavaScript. Обратите внимание, что броузер Netscape 6 и Mozilla, на котором он базируется, больше не поддерживают эти нестандартные применения JavaScript.

12.3. Исполнение программ JavaScript

В предыдущем разделе обсуждались механизмы интеграции кода JavaScript в HTML-файл. Теперь мы обсудим, как именно интегрированный код JavaScript исполняется интерпретатором JavaScript. В следующих разделах поясняется выполнение различных форм кода JavaScript. Несмотря на то что часть этого материала довольно прозрачна, некоторые важные детали не столь очевидны.

12.3.1. Сценарии

Инструкции JavaScript, расположенные между тегами `<script>` и `</script>`, исполняются в порядке их появления; если в файле имеется более одного сценария (тега `<script>`). Если сценарий вызывает `document.write()`, то любой текст, переданный этому методу, вставляется в документ непосредственно после закрывающего тега `</script>` и анализируется HTML-анализатором после завершения исполнения сценария. То же правило применяется к сценариям, включенным из отдельных файлов с помощью атрибута `src`.

Не столь очевидная, но не менее важная деталь, которую стоит запомнить, состоит в том, что исполнение сценария происходит как часть процесса анализа HTML-кода веб-броузером. Поэтому если сценарий присутствует в секции `<head>` HTML-документа, ничего из секции `<body>` еще не определено. Это значит, что объекты JavaScript, представляющие содержимое тела документа, такие как Form и Link, еще не созданы, и этот код не может ими манипулировать.

Сценарии не должны пытаться работать с объектами, которые еще не созданы. Например, нельзя написать сценарий, работающий с содержимым HTML-формы, если он расположен в HTML-файле до этой формы. В каждом отдельном случае применяются другие, схожие правила. Например, есть свойства объекта Document, которые могут быть установлены только из сценария, расположенного в секции `<head>` HTML-документа, до того как броузер начал анализировать содержимое документа в секции `<body>`. Все подобные специальные правила описаны для соответствующего объекта или свойства в справочном разделе, посвященном клиентскому JavaScript.

Сценарии исполняются в процессе анализа и отображения содержащего их HTML-документа, поэтому они не должны работать слишком долго. А поскольку сценарии могут создавать с помощью метода `document.write()` динамическое содержимое, HTML-анализатор должен приостанавливать анализ документа во всех тех случаях, когда интерпретатор JavaScript выполняет сценарий. HTML-документ не может быть целиком выведен, пока все содержащиеся в нем сценарии не завершат свою работу. Если сценарий выполняет какую-либо задачу с интенсивными вычислениями, требующими для завершения много времени, пользователь может быть недоволен ожиданием отображения документа. Поэтому если требуется осуществить в JavaScript много вычислений, необходимо определить функцию, выполняющую эти

вычисления, и вызывать ее из обработчика события, когда пользователь об этом попросит, а не вызывать ее при первой загрузке документа.

Как уже отмечалось, сценарии, использующие атрибут `src` для чтения кода из внешних файлов JavaScript, исполняются точно так же, как те, что включены непосредственно в файл. Это значит, что HTML-анализатор и интерпретатор JavaScript оба останавливаются и ожидают загрузки внешнего файла JavaScript (в отличие от встраиваемых изображений сценарии не могут загружаться в фоновом режиме, пока HTML-анализатор продолжает свою работу). Загрузка внешнего файла с кодом JavaScript, даже по довольно быстрому модемному соединению, может привести к заметным задержкам при загрузке и исполнении веб-страницы. Конечно, как только код JavaScript будет сохранен в локальном кэше, эта неприятность исчезает.

12.3.2. Функции

Помните, что определение функции – это не то же самое, что и ее исполнение. Вы совершенно спокойно можете определить функцию, работающую с объектами, которые еще не созданы. Позаботьтесь только, чтобы функция не выполнялась и не вызывалась, пока не будут существовать все нужные переменные, объекты и т. д. Я уже говорил, что нельзя писать сценарий, работающий с HTML-формой, который находится в HTML-файле перед формой. Однако можно написать сценарий, который определяет функцию, работающую с формой, независимо от относительного местоположения сценария и формы. На самом деле это обычная практика. Многие программы на JavaScript начинаются со сценария в заголовке (`<head>`) документа, который ничего не делает, кроме определения используемых в теле (`<body>`) HTML-файла функций.

Также распространены JavaScript-программы, использующие сценарии просто для того, чтобы определить функции, которые позднее вызываются из обработчиков событий. Как мы увидим в следующем разделе, в этом случае надо позаботиться о двух вещах: во-первых, что все функции определены до того, как какой-либо обработчик события попытается их вызвать, и, во-вторых, что обработчики событий и вызывающие их функции не пытаются использовать объекты, которые еще не определены.

12.3.3. Обработчики событий

Определение обработчика события как значения атрибута `onclick` или другого HTML-атрибута во многом похоже на определение функции JavaScript: обработчик не исполняется сразу. Исполнение обработчиков событий асинхронно. События, как правило, происходят, когда пользователь взаимодействует с HTML-объектами, поэтому нет способа предсказать, когда будет вызван обработчик события.

Обработчики событий разделяют со сценариями важное ограничение: они не должны выполняться долго. Как мы видели, сценарии должны выполняться быстро, поскольку HTML-анализатор не может продолжать, пока сцена-

рий не завершит работу. Обработчики событий также не должны выполняться долго, так как пользователь не сможет взаимодействовать с программой, пока она не закончит обработку события. Если обработчик события выполняет какую-то трудоемкую операцию, пользователю может показаться, что программа зависла или вызвала сбой.

Если по какой-либо причине необходимо выполнить в обработчике операцию, занимающую много времени, убедитесь, что пользователь явно запросил выполнение этой операции, а затем предупредите, что ему придется подождать. Как мы увидим в главе 13, пользователя можно предупредить с помощью диалогового окна `alert()` или вывода текста в строке состояния броузера. Кроме того, если программа требует выполнения большого количества фоновых операций, можно задать повторяющиеся вызовы функции во время простоя с помощью метода `setTimeout()`.

Важно понимать, что обработчики событий могут вызываться до полной загрузки и анализа веб-страницы. Это легко понять, если представить себе медленное сетевое соединение, – даже наполовину загруженный документ может показывать гиперссылки и элементы формы, с которыми может взаимодействовать пользователь, и потому обработчики событий могут быть вызваны до того, как будет загружена вторая половина документа.

Обработчики событий могут вызываться до окончания загрузки документа, и отсюда есть два важных следствия. Во-первых, если ваш обработчик вызывает функцию, вы должны быть уверены, что функция определена до ее вызова. Один из способов обеспечить это состоит в том, чтобы определить все функции в секции `<head>` HTML-документа. Эта секция документа всегда целиком анализируется (и определяются все находящиеся в ней функции) до того, как будет анализироваться секция `<body>`. Поскольку все объекты, определяющие обработчики событий, должны определяться в секции `<body>`, функции, расположенные в секции `<head>`, гарантированно определены до вызова любых обработчиков.

Во-вторых, вы должны быть уверены, что ваш обработчик события не пытается работать с HTML-объектами, которые еще не проанализированы и не созданы. Обработчик, конечно, может всегда спокойно работать с собственным объектом, а также с любыми объектами, определенными в HTML-файле до него. Возможная стратегия состоит в том, чтобы определять пользовательский интерфейс веб-страницы таким образом, чтобы обработчики событий ссылались только на ранее определенные объекты. Например, определяя форму, использующую обработчики событий только в кнопках Submit и Reset, надо просто поместить эти кнопки в нижнюю часть формы (чего и так требует хороший стиль пользовательского интерфейса).

В более сложных программах часто нельзя гарантировать, что обработчики событий работают только с объектами, определенными до них, поэтому в таких программах требуется особая аккуратность. Если обработчик события имеет дело только с объектами, определенными внутри одной формы, у вас вряд ли когда-нибудь возникнут проблемы. Однако когда вы манипулируете объектами в других формах или фреймах, это приобретает серьезное значе-

ние. Один из приемов состоит в том, чтобы проверять существование объекта перед тем, как начинать с ним работать. Это можно сделать путем сравнения его (и любых родительских объектов) с null. Например:

```
<script>
function set_name_other_frame(name)
{
    if (parent.frames[1] == null) return;    // Другой фрейм еще не определен
    if (!parent.frames[1].document) return;  // Документ еще не загружен
    if (!parent.frames[1].document.myform) return;       // Форма еще не определена
    if (!parent.frames[1].document.myform.name) return;  // Поле еще не определено

    parent.frames[1].document.myform.name.value = name;
}
</script>

<input type="text" name="lastname"
       onchange="set_name_other_frame(this.value)";
>
```

В JavaScript 1.5 и более поздних версиях можно опустить проверку существования объектов, если применяется инструкция try/catch для перехвата исключений, генерируемых при вызове функции до полной загрузки документа.

Другой прием, с помощью которого обработчик события может убедиться, что все требуемые объекты определены, требует использования обработчика события onload. Этот обработчик определяется в теге <body> или <frameset> HTML-файла и вызывается, когда документ или набор фреймов полностью загружен. Если в обработчике onload установить флаг, то другие обработчики событий смогут проверить этот флаг, чтобы убедиться в том, что они могут нормально работать, зная, что документ загружен полностью и все содержащиеся в нем объекты определены. Например:

```
<body onload="window.fullyLoaded = true;">
  <form>
    <input type="button" value="Сделай это!"
           onclick="if (window.fullyLoaded) doit();">
  </form>
</body>
```

12.3.3.1. Обработчики событий onload и onunload

Обработчик события onload и его партнер onunload достойны отдельного упоминания в контексте порядка выполнения программ на JavaScript. Оба эти обработчика определяются в теге <body> или <frameset> HTML-файла. (Ни один корректный HTML-файл не может содержать оба этих тега.) Обработчик onload выполняется, когда документ или набор фреймов полностью загружен, то есть когда все изображения загружены и показаны, все подфреймы загружены, все Java-апплеты запущены и т. д. Имейте в виду, что при

работе с несколькими фреймами порядок вызова обработчиков событий `onload` для разных фреймов не определен, за исключением того, что обработчик родительского фрейма вызывается после обработчиков всех своих дочерних фреймов.

Обработчик `onunload` исполняется непосредственно перед выгрузкой страницы, которая происходит, когда броузер собирается перейти на новую страницу. Этот обработчик можно использовать для отмены действий обработчика `onload` или других сценариев вашей веб-страницы. Например, если веб-страница открывает вторичное окно броузера, обработчик `onunload` предоставляет возможность закрыть это окно при переходе пользователя к какой-либо другой веб-странице. Обработчик `onunload` не должен выполнять каких-либо трудоемких действий и не должен открывать диалоговых окон. Он существует просто для выполнения быстрых операций очистки; его работа не должна замедлять работу или препятствовать переходу пользователя к новой странице.

12.3.4. URL JavaScript

Код JavaScript в URL `javascript:` не выполняется во время загрузки документа, содержащего этот URL. Он не интерпретируется, пока броузер не попытается загрузить документ, на который ссылается URL. Это может происходить, когда пользователь вводит URL JavaScript или, более часто, когда пользователь следует по ссылке, щелкает по клиентской карте изображений или передает данные формы. URL `javascript:` часто используется как альтернатива обработчикам событий и, как и в обработчиках, код в этих URL может исполняться до полной загрузки документа. Поэтому необходимо соблюдать те же предосторожности с URL `javascript:`, которые соблюдаются в обработчиках событий, чтобы гарантировать, что они не пытаются ссылаться на еще не определенные объекты (или функции).

12.3.5. Время жизни окон и переменных

Последняя тема в нашем исследовании работы клиентских программ в JavaScript – это время жизни переменных. Мы видели, что объект Window – это глобальный объект для клиентского JavaScript и что все глобальные переменные представляют собой свойства объекта Window. Что происходит с объектами Window и содержащимися в нем переменными, когда веб-броузер переходит от одной веб-страницы к другой?

При загрузке в окно или фрейм нового документа объект Window для этого окна возвращается в свое состояние по умолчанию: удаляются любые свойства и функции, определенные сценарием в предыдущем документе, и восстанавливаются все стандартные системные свойства, которые были изменены или переписаны. Любой документ начинается с чистого листа. Сценарии могут использовать этот факт, – они не будут наследовать нарушенную предыдущим документом среду. Любые переменные и функции, определяемые сценарием, сохраняются только пока документ не будет заменен на новый.

Чистый лист, который мы здесь обсуждаем, – это объект Window, представляющий окно или фрейм, в который загружается документ. Как мы уже говорили, объект Window – это глобальный объект для кода JavaScript в этом окне или фрейме. Однако если вы работаете с несколькими фреймами или окнами, сценарий в одном окне может ссылаться на объекты Window, представляющие другие окна или фреймы. Поэтому кроме рассмотрения постоянства переменных и функций, определенных в объектах Window, мы также должны рассмотреть постоянство самого объекта Window.

Объект Window, представляющий окно броузера верхнего уровня, существует, пока существует это окно. Ссылка на объект Window остается действующей независимо от того, сколько веб-страниц было загружено в окно и выгружено из него. Объект Window действует до тех пор, пока открыто окно верхнего уровня.[1]

Объект Window, представляющий фрейм, остается действующим, пока этот фрейм сохраняется внутри фрейма или окна, в котором он содержится. Например, если фрейм A содержит сценарий, в котором имеется ссылка на объект Window фрейма B, и во фрейм B загружается новый документ, присутствующая во фрейме A ссылка на объект Window остается действующей. Когда будет загружен новый документ, любые переменные или функции, определенные в объекте Window фрейма B, будут удалены, но сам объект Window остается действующим (пока содержащий их фрейм или окно не загрузит новый документ и не перепишет как фрейм A, так и фрейм B).

Это значит, что объекты Window, представляющие как окна верхнего уровня, так и фреймы, достаточно постоянны. Время жизни объекта Window может намного превышать время жизни содержащихся и отображаемых в нем веб-страниц, равно как и время жизни сценариев, содержащихся в этих веб-страницах.

---

[1] Объект Window может не удаляться фактически, когда закрывается это окно. Если на этот объект Window все еще существуют ссылки из других окон, объект не удаляется сборщиком мусора. Однако ссылка на окно, которое уже закрыто, не представляет практического интереса.

13

Окна и фреймы

В главе 12 был описан объект Window и центральная роль, которую он играет в клиентском JavaScript. Мы видели, что объект Window – глобальный для программ клиентского JavaScript и, как это было проиллюстрировано на рис. 12.1, также представляет собой корень иерархии клиентских объектов.

Кроме этих особых ролей, объект Window важен сам по себе. Каждое окно веб-броузера и каждый фрейм внутри каждого окна представлены объектом Window. Объект Window определяет довольно много свойств и методов, важных для программирования на клиентском JavaScript. В данной главе рассматриваются эти свойства и методы и демонстрируется несколько важных приемов программирования окон и фреймов. Заметьте, что поскольку объект Window играет столь важную роль в клиентском программировании, эта глава довольно велика. Не чувствуйте себя обязанным освоить весь этот материал сразу, быть может, вам будет легче изучить его по частям.

13.1. Обзор объекта Window

Начнем с обзора некоторых наиболее часто используемых свойств и методов объекта Window. В последующих разделах главы этот материал рассматривается более подробно. А полное описание свойств и методов объекта Window можно найти в части IV «Справочник по клиентскому JavaScript».

Вот наиболее важные свойства объекта Window:

`closed`

Логическое значение, равное `true`, только если окно было закрыто.

`defaultStatus`, `status`

Текст, отображаемый в строке состояния броузера.

`document`

Ссылка на объект Document, представляющий HTML-документ, отображаемый в окне. Объект Document подробно рассматривается в главе 14.

`frames[]`

Массив объектов Window, представляющих фреймы (если они имеются) внутри окна.

`history`

Ссылка на объект History, представляющий историю просмотра страниц пользователем в этом окне.

`location`

Ссылка на объект Location, представляющий URL документа, который отображается в окне. Если это свойство установлено, то загружается новый документ.

`name`

Имя окна. Может, например, применяться с атрибутом `target` тега `<a>`.

`opener`

Ссылка на объект Window, открывший данное окно, или `null`, если это окно было открыто пользователем.

`parent`

Если текущее окно – это фрейм, то в данном свойстве содержится ссылка на фрейм окна, содержащий данный фрейм.

`self`

Ссылка на самого себя, на текущий объект Window. Синоним `window`.

`top`

Если текущее окно – фрейм, в данном свойстве содержится ссылка на объект Window окна верхнего уровня, содержащего данный фрейм. Обратите внимание, что для вложенных фреймов `top` отличается от `parent`.

`window`

Ссылка на самого себя, на текущий объект Window. Синоним `self`.

Объект Window также поддерживает несколько важных методов:

`alert()`, `confirm()`, `prompt()`

Отображают простые диалоговые окна, кроме того, `confirm()` и `prompt()`, получают отклик от пользователя.

`close()`

Закрывает окно.

`focus()`, `blur()`

Запрашивает фокус клавиатурного ввода для окна или отказывается от него. Метод `focus()` также обеспечивает видимость окна, выводя его в начало стека окон.

`moveBy()`, `moveTo()`

Перемещает окно.

`open()`

Открывает новое окно верхнего уровня для отображения указанного URL с заданным набором возможностей.

`print()`

Печатает окно или фрейм – то же самое, что происходит при щелчке по кнопке Print из панели инструментов текущего окна (только Netscape 4, IE 5 и более поздние версии).

`resizeBy()`, `resizeTo()`

Изменение размеров окна.

`scrollBy()`, `scrollTo()`

Прокрутка документа, отображаемого внутри окна.

`setInterval()`, `clearInterval()`

Планирование или отмена повторного вызова функции с указанной задержкой между вызовами.

`setTimeout()`, `clearTimeout()`

Планирование и отмена вызова функции через указанное количество миллисекунд.

Как можно видеть из этих списков, объект Window предоставляет довольно богатую функциональность. Далее в этой главе значительная ее часть изучается подробнее.

13.2. Простые диалоговые окна

Чаще других применяются три метода – `alert()`, `confirm()` и `prompt()`. Они выводят простые диалоговые окна. Метод `alert()` выводит сообщение пользователю, метод `confirm()` просит пользователя щелкнуть по кнопке Ok или Cancel для подтверждения или отмены операции, а `prompt()` просит пользователя ввести строку. Примеры диалоговых окон, выводимых этими тремя методами, показаны на рис. 13.1.

Обратите внимание, что в этих окнах отображается обычный текст, а не HTML. Его можно форматировать только пробелами, переводами строк и различными знаками пунктуации. Требуемое размещение обычно достигается методом проб и ошибок. Однако имейте в виду, что диалоговые окна выглядят по-разному на разных платформах и в разных броузерах, поэтому не всегда можно рассчитывать, что текст будет выглядеть правильно во всех возможных броузерах.

Некоторые броузеры (такие как Netscape 3 и 4) отображают слово «JavaScript» в заголовке или верхнем левом углу всех диалоговых окон, создаваемых методами `alert()`, `confirm()` и `prompt()`. Хотя дизайнеров этот факт

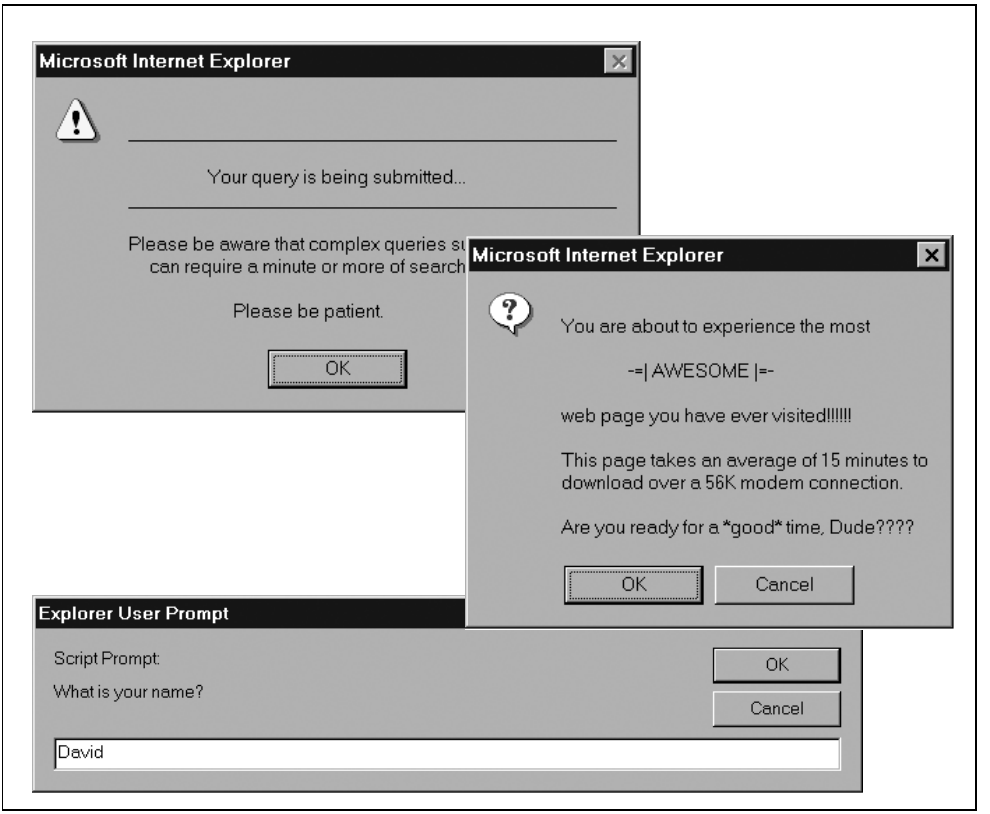

Рис. 13.1. *Диалоговые окна alert( ), confirm( ) и prompt( )*

раздражает, его стоит рассматривать как особенность, а не как ошибку: слово «JavaScript» находится там, чтобы пользователю было ясно происхождение диалогового окна и чтобы помешать созданию кода «троянских коней», имитирующих системные диалоговые окна и обманом заставляющих пользователей вводить свои пароли и выполнять другие действия, которые им выполнять не следует.

Методы `confirm()` и `prompt()` являются *блокирующими*, то есть они не возвращают управление, пока пользователь не закроет отображаемые ими диалоговые окна. Это значит, что когда выводится одно из этих окон, код прекращает выполнение, и текущий загружаемый документ, если таковой существует, прекращает загрузку до тех пор, пока пользователь не отреагирует на запрос. Такому поведению методов нет альтернативы: возвращаемое ими значение – это введенные пользователем данные, поэтому они просто должны дождаться реакции пользователя перед тем, как вернуть значение. В большинстве броузеров метод `alert()` также является блокирующим и ожидает от пользователя закрытия диалогового окна. Однако в некоторых броузерах (в частности, в Netscape 3 и 4 на Unix-платформах) `alert()` не является

блокирующим. На практике эта небольшая несовместимость редко создает проблемы.

Несколько типичных вариантов применения этих методов показано ниже, в примере 13.1.

Пример 13.1. Методы alert( ), confirm( ) и prompt( )

```
// Это функция, использующая метод alert() для сообщения пользователю
// о том, что сохранение формы потребует определенного времени, и что пользователь
// должен иметь терпение. Она подходит для использования в обработчике события
// onsubmit в HTML-форме.
// Обратите внимание, что все форматирование выполняется с помощью пробелов,
// переводов строк и подчеркиваний.
function warn_on_submit()
{
    alert("\n_____________________________________________________\n\n" +
          "                          Ваш запрос был сохранен...\n"    +
          "_____________________________________________________\n\n"   +
          "Имейте в виду, такие сложные вопросы как ваш\n"            +
          "могут потребовать одной или нескольких минут поиска.\n\n" +
          "                          Просьба подождать.");
}

// Это применение метода confirm(), спрашивающего, действительно ли пользователь
// хочет посетить веб-страницу, которая долго загружается.
// Обратите внимание, что возвращаемое методом значение обозначает ответ
// пользователя. На основе этого ответа мы перенаправляем броузер на
// нужную страницу.
var msg = "\nСейчас вы сможете увидеть самую \n\n" +
          "                 -=| УЖАСНУЮ |=-\n\n" +
          "веб-страницу, из тех, что вы когда-либо посещали!!!!!\n\n" +
          "Загрузка страницы занимает в среднем 15 минут для\n" +
          " модемного соединения со скоростью 56K.\n\n" +
          "Ты готов????";

if (confirm(msg))
    location.replace("awesome_page.html");
Else
    location.replace("lame_page.html");

// Вот немного простого кода, посредством метода prompt()получающего
// имя пользователя, а затем вставляющего это имя в динамически генерируемый
// HTML-код.
n = prompt("Ваше имя?", "");
document.write("<hr><h1>Добро пожаловать на мою домашнюю страницу, " + n +
"</h1><hr>");
```

13.3. Строка состояния

Веб-броузеры обычно отображают в нижней части любого окна *строку состояния* (кроме тех окон, которые явно созданы без нее), предназначенную для вывода сообщений пользователю. Когда пользователь, например, перемещает указатель мыши над гиперссылкой, броузер обычно показывает URL, на который эта ссылка указывает. А когда пользователь перемещает указатель над кнопкой броузера, в строке состояния выводится контекстная подсказка, разъясняющая ее назначение. Строку состояния можно использовать в собственных программах. Ее содержимое управляется двумя свойствами объекта Window: `status` и `defaultStatus`.

Обычно, когда пользователь проводит указателем мыши над ссылкой, веб-броузеры отображают URL этой ссылки, но встречаются ссылки, которые ведут себя по-другому, – они отображают вместо URL некоторый текст. Этот эффект достигается с помощью свойства `status` объекта Window и обработчика событий `onmouseover` данной гиперссылки:

```
<!-- Так устанавливается строка статуса в гиперссылке.
  -- Обратите внимание, что обработчик события *должен* для того,
  -- чтобы это работало, возвращать true -->
Заблудились? Запутались? Посетите
<a href="sitemap.html" onmouseover="status='Переход к карте сайта'; return true;">
  Карту Сайта
</a>

<!-- Вы можете сделать то же самое с помощью клиентских карт изображений -->
<img src="images/imgmap1.gif" usemap="#map1">
<map name="map1">
  <area coords="0,0,50,20" href="info.html"
    onmouseover="status='Посетите наш информационный центр'; return true;">
  <area coords="0,20,50,40" href="order.html"
    onmouseover="status='Сделайте заказ'; return true;">
  <area coords="0,40,50,60" href="help.html"
    onmouseover="status='Быстро получите помощь!'; return true;">
</map>
```

Обработчик события `onmouseover` в этом примере должен возвращать `true`. Это указание для броузера о том, что он не должен выполнять собственное действие по умолчанию для данного события, то есть не должен отображать URL ссылки в строке состояния. Если не вернуть `true`, броузер перепишет любое сообщение, выведенное разработчиком в строку состояния, своим собственным сообщением с URL. Не беспокойтесь, если пока не совсем понимаете, как функционирует обработчик в этом примере. Описание событий ожидает вас в главе 19.

Когда пользователь перемещает указатель мыши над гиперссылкой, броузер отображает URL ссылки, а затем удаляет URL из строки состояния, когда указатель уходит от нее. То же самое происходит, когда обработчик события `onmouseover` используется для установки свойства `status` объекта Window, –

ваше сообщение отображается, пока указатель мыши находится над гиперссылкой, и удаляется, когда указатель мыши уходит с нее.

Свойство status предназначено именно для таких временных сообщений, которые мы видели в предыдущем примере. Однако иногда в строке состояния требуется показать постоянное сообщение, например приветствие пользователям, посетившим вашу страницу, или подсказку для новых посетителей. Чтобы это сделать, установите свойство defaultStatus объекта Window, задающее текст, отображаемый в строке состояния по умолчанию. Этот текст временно заменяется URL, контекстными подсказками или другими временными сообщениями при нахождении указателя мыши над гиперссылками или кнопками броузера, но как только указатель мыши уходит из этих областей, текст по умолчанию восстанавливается.

Посредством свойства defaultStatus можно вывести дружественное и полезное сообщение для самых начинающих:

```
<script>
defaultStatus = "Добро пожаловать!  Для навигации нажмите на подчеркнутый синий
текст.";
</script>
```

13.4. Время запуска и интервалы

Метод setTimeout() объекта Window планирует запуск фрагмента кода JavaScript на определенный момент времени в будущем. Метод clearTimeout() может применяться для отмены исполнения такого кода. setTimeout() чаще всего служит для выполнения анимации и других видов повторяющихся действий. Если функция запускается, а затем вызывает setTimeout() для планирования повторного своего вызова, мы получаем процесс, повторяющийся без вмешательства пользователя. JavaScript 1.2 добавляет методы setInterval() и clearInterval(), которые похожи на setTimeout() и clearTimeout() за исключением того, что они автоматически заново планируют повторное выполнение; код не должен планировать свое повторное выполнение сам.

Метод setTimeout() чаще всего применяется в сочетании со свойствами status или defaultStatus для вывода какого-либо сообщения в строке состояния броузера. Как правило, анимация в строке состояния получается безвкусной, и вам следует избегать ее! Однако некоторые приемы анимации в строке состояния могут быть полезны и соответствовать хорошему вкусу. Именно такая, сделанная со вкусом анимация в строке состояния, показана в примере 13.2. Она отображает там текущее время и обновляет это время раз в минуту. Обновление происходит всего раз в минуту, поэтому анимация не отвлекает пользователя постоянным миганием в нижней части окна броузера, как это делают многие другие.

Обратите внимание на применение обработчика события onload тега <body> для выполнения первого вызова метода display_time_in_status_line(). Этот обработчик вызывается один раз, когда HTML-документ полностью загру-

жен в броузер. После этого первого вызова метод использует setTimeout() для планирования своего вызова каждые 60 секунд, чтобы он мог обновлять отображаемое время.

Пример 13.2. Цифровые часы в строке состояния

```
<html>
<head>
<script>
// Эта функция отображает время в строке состояния
// Вызовите ее один раз для активизации часов, далее она будет вызываться
// самостоятельно
function display_time_in_status_line()
{
    var d = new Date();                 // Получаем текущее время
    var h = d.getHours();               // Количество часов: от 0 до 23
    var m = d.getMinutes();             // Количество минут: от 0 до 59
    var ampm = (h >= 12)?"PM":"AM";     // a.m. или p.m.?
    if (h > 12) h -= 12;               // Преобразуем 24-часовой формат в 12-часовой
    if (h == 0) h = 12;                 // Преобразуем 0 в полночь
    if (m < 10) m = "0" + m;            // Преобразуем 0 минут в 00 минут и т.д.
    var t = h + ':' + m + ' ' + ampm;  // Выводим все это

    defaultStatus = t;                  // Выводим время в строке состояния

    // Устанавливаем повторный запуск через одну минуту
    setTimeout("display_time_in_status_line()", 60000);  // 60000 мс -
}                                                        // это одна минута
</script>
</head>
<!-- Не занимайтесь загрузкой часов, пока не будет загружено все остальное.
  -- Все равно во время загрузки строка состояния будет занята другими
  -- сообщениями -->
<body onload="display_time_in_status_line();">
<!-- Здесь располагается содержимое HTML-документа -->
</body>
</html>
```

В JavaScript 1.2 пример 13.2 мог бы быть записан с применением метода setInterval() вместо setTimeout(). В этом случае должен быть удален вызов setTimeout()из метода display_time_in_status_line() и обработчик события onload. Вместо этого после определения функции display_time_in_status_line() сценарий должен вызвать метод setInterval() для задания автоматически повторяющегося через каждые 60 000 миллисекунд вызова функции.

13.5. Обработка ошибок

Свойство onerror объекта Window – это особенный обработчик. Если присвоить этому свойству функцию, она будет вызываться во всех тех случаях, ког-

да в этом окне возникает ошибка: присвоенная этому свойству функция становится обработчиком ошибок для этого окна.

Обработчику ошибок передается три аргумента. Первый аргумент – это сообщение, описывающее произошедшую ошибку. Это может быть что-то вроде «отсутствует оператор в выражении», «self доступно только для чтения» или «myname не определен». Второй аргумент – это строка, содержащая URL документа с кодом JavaScript, приведшим к ошибке. Третий аргумент – это номер строки в документе, где произошла ошибка. Обработчик ошибок может использовать эти аргументы для любых своих целей. Типичный обработчик ошибок может показать сообщение пользователю, записать его в протокол или потребовать игнорирования ошибки.

Кроме этих трех аргументов важно значение, возвращаемое обработчиком `onerror`. Броузеры в случае возникновения ошибки обычно отображают сообщение в диалоговом окне или строке состояния. Если обработчик `onerror` возвращает `true`, это говорит системе о том, что ошибка обработана и никаких дальнейших действий не требуется; другими словами, система не должна выводить собственное сообщение об ошибке. Например, если вы независимо от количества ошибок в написанном в вами коде не хотите, чтобы ваших пользователей донимали сообщения об ошибках, можете поместить следующую строку кода в начало всех своих программ на JavaScript:

```
self.onerror = function() { return true; }
```

Сделав это, вы, конечно, сильно усложните пользователям осуществление обратной связи с вами при молчаливых сбоях в программе, не выдающей сообщений об ошибках.

Применение `onerror` мы увидим в примере 14.1. В нем обработчик `onerror` отображает подробное сообщение об ошибке, и пользователь может создать и отправить отчет об ошибках, содержащий все эти подробности.

Заметим, что в Netscape 6 обработчик ошибок `onerror` имеет ошибку. Хотя заданная вами функция и вызывается при возникновении ошибки, переданные ей три аргумента некорректны и непригодны для использования. Однако Netscape 6 и другие броузеры, поддерживающие JavaScript 1.5, имеют альтернативные средства для перехвата и обработки ошибок: они могут обратиться к инструкции `try/catch` (подробности см. в главе 6).

13.6. Объект Navigator

Свойство `Window.navigator` ссылается на объект Navigator, содержащий общую информацию о веб-броузере, такую как версия и список форматов данных, которые он может отображать. Объект Navigator назван в честь Netscape Navigator, но он также поддерживается в Internet Explorer. IE также поддерживает `clientInformation` как нейтральный синоним для `navigator`. К сожалению, Netscape и Mozilla не поддерживают это свойство.

Объект Navigator имеет пять свойств, предоставляющих информацию о версии работающего броузера:

appName

Название веб-броузера.

appVersion

Номер версии и/или другая информация о версии броузера. Обратите внимание, что его следует рассматривать как внутренний номер версии, поскольку он не всегда соответствует номеру, отображаемому для пользователя. Так, Netscape 6 сообщает о себе номер версии 5.0, хотя версии Netscape 5 никогда не было. Кроме того, все версии IE от 4 до 6 сообщают о себе номер версии 4.0, который указывает на совместимость с базовой функциональностью броузеров 4-го поколения.

userAgent

Строка, которую броузер посылает в HTTP-заголовке USER-AGENT. Это свойство обычно содержит всю ту информацию, которая содержится в свойствах appName и appVersion.

appCodeName

Кодовое имя броузера. Для Netscape значение кодового имени установлено в «Mozilla». Для совместимости IE делает то же самое.

platform

Аппаратная платформа, на которой работает броузер. Это свойство было добавлено в JavaScript 1.2.

Следующие строки кода JavaScript выводят все эти свойства объекта Navigator в диалоговом окне.

```
var browser = "BROWSER INFORMATION:\n";
for(var propname in navigator) {
    browser += propname + ": " + navigator[propname] + "\n"
}
alert(browser);
```

Диалоговое окно, представленное на рис. 13.2, выводится при запуске этого кода в IE 6.

Из этого рисунка видно, что свойства объекта Navigator иногда содержат более сложную информацию, чем та, которая нас интересует. Например, часто достаточно знать лишь первые цифры из свойства appVersion. Для извлечения из объекта Navigator только необходимой информации о броузере часто применяются методы parseInt() и String.indexOf(). В примере 13.3 показан код, обрабатывающий свойства объекта Navigator и сохраняющий их в объекте с именем browser. С обработанными свойствами иметь дело проще, чем с исходными значениями свойств объекта navigator. Общий термин для такого кода – «клиентский снифер», и в Интернете можно найти код более слож-

ных и универсальных сниферов.[1] Однако для многих целей прекрасно работают и такие простые фрагменты кода.

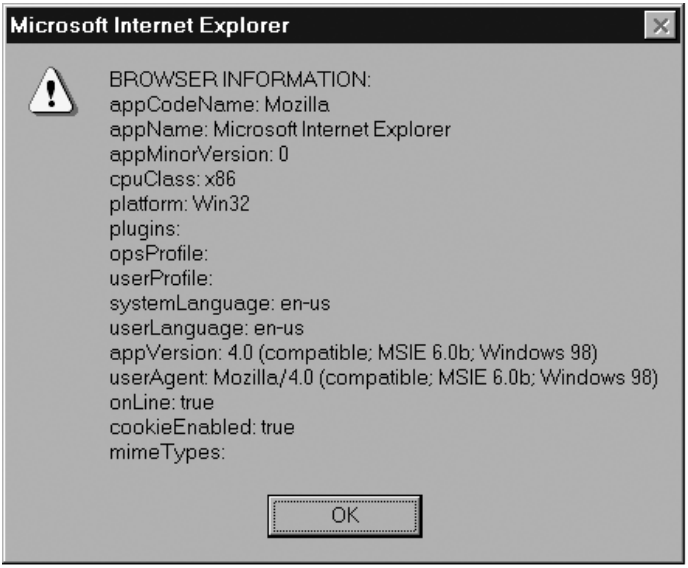

Рис. 13.2. *Свойства объекта Navigator*

Пример 13.3. Определение разработчика и номера версии броузера

```
/*
 * File: browser.js
 * Include with: <script SRC="browser.js"></script>
 *
 * Простой "sniffer", определяющий разработчика и номер версии броузера.
 * Он создает объект browser, который проще в использовании,
 * чем объект navigator.
 */
// Создание объекта browser
var browser = new Object();

// Определение основной версии броузера
browser.version = parseInt(navigator.appVersion);
// Теперь, выясняем, относится ли броузер к одному из основных.
// Начинаем с отрицательного предположения.
browser.isNetscape = false;
browser.isMicrosoft = false;
if (navigator.appName.indexOf("Netscape") != -1)
    browser.isNetscape = true;
else if (navigator.appName.indexOf("Microsoft") != -1)
    browser.isMicrosoft = true;
```

[1] Например, *http://www.mozilla.org/docs/web-developer/sniffer/browser_type.html*.

13.7. Объект Screen

В JavaScript 1.2 свойство screen объекта Window ссылается на объект Screen, предоставляющий информацию о размере экрана пользователя и доступного количества цветов. Свойства width и height задают размер экрана в пикселах. Свойства availWidth и availHeight задают реально доступный размер экрана; из них исключается пространство, требуемое для таких графических элементов, как панель задач Windows. Эти свойства позволяют, например, определять размер включаемых в документ изображений, или окон, создаваемых в программе, использующей несколько окон броузера.

Свойство colorDepth задает логарифм по основанию 2 от количества отображаемых цветов. Часто это значение совпадает с количеством битов, отведенных для каждого пиксела изображения. Так, 8-битовый дисплей может отображать 256 цветов, и если все эти цвета доступны для броузера, свойство screen.colorDepth должно быть равно 8. Однако иногда броузер ограничивает себя подмножеством доступных цветов, и значение screen.colorDepth будет меньше количества битов на пиксел дисплея. При наличии нескольких версий изображения, созданных с разным количеством цветов, это свойство позволяет определить, какую версию включать в документ.

Работу с объектом Screen иллюстрирует пример 13.4 далее в этой главе.

13.8. Методы управления окнами

Объект Window определяет несколько методов, предназначенных для высокоуровневого управления самим окном. В следующих разделах рассматривается, как эти методы позволяют нам открывать и закрывать окна, управлять их положением и размером, запрашивать и отдавать фокус ввода и прокручивать содержимое окна. Закончим мы примером, демонстрирующим некоторые из этих возможностей.

13.8.1. Открытие окон

Для открытия нового окна броузера предназначен метод open() объекта Window. Этот метод принимает четыре необязательных аргумента и возвращает объект Window, представляющий только что открытое окно. Первый аргумент open() – это URL документа, отображаемого в новом окне. Если этот аргумент отсутствует (или равен null или пустой строке), окно будет пустым.

Второй аргумент open() – это имя окна. Как мы обсудим позднее в данной главе, это имя может быть полезно в качестве значения атрибута target тега <form> или <a>. Если указать имя уже существующего окна, open() вернет ссылку на это существующее окно, а не откроет новое.

Третий необязательный аргумент open() – это список параметров, задающих размер и элементы графического пользовательского интерфейса окна. Если опустить этот аргумент, окно получает размер по умолчанию и имеет пол-

ный набор графических элементов: меню, строка состояния, панель инструментов и т. д. Указав этот аргумент, можно явно задать размер окна и набор имеющихся в нем элементов управления. Например, маленькое окно с изменяемым размером, имеющее строку состояния, но не содержащее меню, панели инструментов и поле Location, можно открыть посредством следующей строки кода JavaScript:

```
var w = window.open("smallwin.html", "smallwin",
                    "width=400,height=350,status=yes,resizable=yes");
```

Обратите внимание, что когда указывается третий аргумент, любые не указанные явно элементы управления отсутствуют. Полный набор доступных элементов и их имен приведен в описании метода `Window.open()` в части IV «Справочник по клиентскому JavaScript».

Четвертый аргумент `open()` полезен, только если второй аргумент представляет собой имя существующего окна. Этот, четвертый аргумент – логическое значение, определяющее, должен ли URL, указанный в первом аргументе, заменять текущую запись в истории просмотра окна (`true`), или должна быть создана новая запись (`false`). Последний вариант выбирается по умолчанию.

Возвращаемое методом `open()` значение является объектом Window, представляющим только что созданное окно. Этот объект в коде JavaScript позволяет сослаться на новое окно так же, как объект `window` – на окно, в котором работает ваш код. А как насчет обратной ситуации? Что если код JavaScript в новом окне хочет обращаться к открывшему его окну? В JavaScript 1.1 и более поздних версиях свойство `opener` объекта Window ссылается на окно, из которого было открыто текущее окно. Если окно было создано пользователем, а не кодом JavaScript, значение свойства `opener` равно `null`.

Говоря о методе `open()`, важно отметить, что он почти всегда вызывается как `window.open()`, даже несмотря на то что `window` ссылается на глобальный объект и, следовательно, его указание не обязательно. Объект `window` задается явно, поскольку объект Document также имеет метод `open()`, поэтому `window.open()` позволяет абсолютно точно указать, что мы пытаемся сделать. Это не просто полезная привычка; в некоторых ситуациях это обязательно, потому что, как мы узнаем в главе 19, обработчики событий выполняются в контексте того объекта, в котором они определены. Когда, например, выполняется обработчик события HTML-кнопки, цепочка областей видимости включает объект Button, объект Form, содержащий эту кнопку, объект Document, содержащий форму, и, наконец, объект Window, содержащий документ. Поэтому если такой обработчик ссылается просто на метод `open()`, этот идентификатор разрешается в итоге в объекте Document, и обработчик открывает новый документ, а не новое окно!

Практическое применение метода `open()` иллюстрируется примером 13.4.

13.8.2. Закрытие окон

Новое окно открывается при помощи метода `open()`, а при помощи метода `close()` – закрывается. Если мы создали объект Window `w`, то закрыть его можно инструкцией:

```
w.close();
```

Код JavaScript, работающий внутри данного окна, может закрыть его так:

```
window.close();
```

Снова обратите внимание на явное использование идентификатора `window` для устранения неоднозначности между методом `close()` объекта Window и методом `close()` объекта Document.

Большинство броузеров разрешают программисту автоматически закрывать только те окна, которые были созданы его собственным кодом JavaScript. Если он попытается закрыть любое другое окно, будет выведено диалоговое окно, спрашивающее у пользователя подтверждение (или отмену) запроса на закрытие окна. Эта предосторожность не дает неосмотрительным создателям сценариев писать код, закрывающий главное окно броузера пользователя.

В JavaScript 1.1 и более поздних версиях объект Window продолжает существовать и после закрытия представляемого им окна. Однако не следует использовать какие-либо его свойства или методы, кроме проверки свойства `closed`. Это свойство равно `true`, если окно было закрыто. Помните, что пользователь может закрывать любые окна в любое время, поэтому, чтобы избежать ошибок, полезно периодически проверять, открыто ли окно, с которым вы пытаетесь работать. Как это делается, показано в примере 13.4.

13.8.3. Геометрия окна

В JavaScript 1.2 метод `moveTo()` перемещает левый верхний угол окна в точку с указанными координатами. Похожим образом `moveBy()` перемещает окно на указанное количество пикселов влево или вправо и вверх или вниз. Методы `resizeTo()` и `resizeBy()` изменяют размер окна на абсолютное или относительное значение; они тоже впервые появились в JavaScript 1.2. Обратите внимание, что для предотвращения атак на систему безопасности, основанных на коде, работающем в маленьких или расположенных за пределами экрана окнах, которые пользователь не замечает, броузер может ограничить вашу способность перемещения окон за пределы экрана и сильного уменьшения их размера.

13.8.4. Фокус ввода и видимость

Методы `focus()` и `blur()` также предоставляют средства высокоуровневого управления окном. Вызов `focus()` запрашивает от системы фокус ввода для окна, а `blur()` освобождает фокус. Кроме того, метод `focus()` гарантирует, что

окно будет видимым, перенося его в начало стека окон. Когда новое окно открывается с помощью метода `Window.open()`, броузер автоматически создает окно в начале стека окон. Но если второй аргумент задает имя уже существующего окна, метод `open()` не делает автоматически окно видимым. Поэтому часто за вызовами `open()` следует вызов `focus()`.

Методы `focus()` и `blur()` определены в JavaScript 1.1 и более поздних версиях.

13.8.5. Прокрутка

Объект Window также содержит методы, прокручивающие документ внутри окна или фрейма. Метод `scrollBy()` прокручивает документ на указанное количество пикселов влево или вправо и вверх или вниз, а `scrollTo()` прокручивает документ на абсолютную позицию. Он перемещает документ таким образом, что точка документа с указанными координатами отображается в левом верхнем углу области документа в окне. Эти два метода определены в JavaScript 1.2. В JavaScript 1.1 метод `scroll()` выполняет ту же функцию, что и метод `scrollTo()` в JavaScript 1.2. Метод `scrollTo()` – это предпочтительный вариант вызова метода, но `scroll()` сохранен для обратной совместимости.

В JavaScript 1.2 элементами массива `anchors[]` объекта Document являются объекты Anchor. Каждый объект Anchor имеет свойства `x` и `y`, задающие якоря внутри документа. Следовательно, эти свойства в сочетании с методом `scrollTo()` позволяют осуществлять прокрутку к известным местам в документе. В качестве альтернативы в IE 4 и более поздних версиях, а также в Netscape 6 и более поздних версиях все элементы документа определяют метод `focus()`. Вызов этого метода для элемента приводит к прокрутке документа для обеспечения видимости данного элемента.

13.8.6. Пример применения методов объекта Window

Пример 13.4 демонстрирует методы `open()`, `close()` и `moveTo()` объекта Window, а также некоторые другие обсуждавшиеся нами приемы работы с окнами. В примере создается новое окно и затем с помощью метода `setInterval()` задаются интервалы повторяющихся вызовов функции, перемещающей это окно по экрану. Размер экрана определяется с помощью объекта Screen и потом на основе этих данных осуществляется отскок окна при достижении им любого края экрана.

Пример 13.4. Перемещение окна

```
<script>
// Это начальные значения для нашей анимации.
var x = 0, y = 0, w=200, h=200;  // Позиция и размер окна
var dx = 5, dy = 5;              // Скорость перемещения окна
var interval = 100;              // Количество миллисекунд между обновлениями

// Создаем окно, которое мы собираемся перемещать по экрану.
```

```
// URL javascript: - это простой способ показа короткого документа
// Последний аргумент задает размер окна.
var win = window.open('javascript:"<h1>ОТСКОК!</h1>"', "",
                      "width=" + w + ",height=" + h);

// Установка начальной позиции окна
win.moveTo(x,y);

// Используйте setInterval() для вызова метода bounce() через количество
// миллисекунд, указанных в interval. Сохраняем возвращаемое значение,
// чтобы мы могли остановить анимацию, передав значение в clearInterval().
var intervalID  = window.setInterval("bounce()", interval);

// Эта функция перемещает окно на (dx, dy) каждые interval миллисекунд
// Когда окно достигает края экрана, оно осткакивает.
function bounce() {
    // Останавливаем анимацию, если окно закрыто
    if (win.closed) {
        clearInterval(intervalID);
        return;
    }

    // Отскакивает, если достигнут правый или левый край
    if ((x+dx > (screen.availWidth - w)) || (x+dx < 0)) dx = -dx;

    // Отскакивает, если достигнут нижний или верхний край
    if ((y+dy > (screen.availHeight - h)) || (y+dy < 0)) dy = -dy;

    // Обновляем текущее положение окна
    x += dx;
    y += dy;

    // И наконец, перемещаем окно в новую позицию.
    win.moveTo(x,y);
}
</script>

<!-- Нажатие на эту кнопку прекращает анимацию! -->
<form>
<input type="button" value="Stop"
       onclick="clearInterval(intervalID); win.close();">
</form>
```

13.9. Объект Location

Свойство `location` окна, являющееся ссылкой на объект Location, представляет URL документа, отображаемого в данный момент в текущем окне. Свойство `href` объекта Location – это строка, содержащая полный текст URL. Другие свойства этого объекта, такие как `protocol`, `host`, `pathname` и `search`, задают отдельные части URL.

Свойство search объекта Location представляет особый интерес. Оно содержит любую часть URL, следующую за вопросительным знаком (включая сам знак вопроса). Обычно это что-то вроде строки запроса. Вопросительный знак в URL – это прием для встраивания аргументов в URL. Хотя эти аргументы обычно предназначены для сценариев CGI, работающих на сервере, нет причины, по которой они не могут также использоваться в страницах, содержащих JavaScript. Пример 13.5 показывает определение универсальной функции getArgs(), позволяющей извлекать аргументы из свойства search URL. Пример также показывает, как этот метод мог бы применяться для установки начальных параметров анимации окна в примере 13.4.

Пример 13.5. Извлечение аргументов из URL

```
/*
 * Эта функция выделяет разделенные запятыми пары аргументов name=value
 * из строки запроса в URL. Она сохраняет эти пары в свойствах объекта
 * и возвращает этот объект.
 */
function getArgs() {
    var args = new Object();
    var query = location.search.substring(1);     // Получает строку запроса
    var pairs = query.split(",");                 // Разбиение по запятым
    for(var i = 0; i < pairs.length; i++) {
        var pos = pairs[i].indexOf('=');          // Поиск пар "name=value"
        if (pos == -1) continue;                  // Если не найдено, пропускаем
        var argname = pairs[i].substring(0,pos);  // Извлекаем имя
        var value = pairs[i].substring(pos+1);    // Извлекаем значение
        args[argname] = unescape(value);          // Сохраняем его в свойстве
        // В JavaScript 1.5 используйте вместо unescape()
    }   // метод decodeURIComponent()
    return args;                                  // Возвращаем объект
}

/*
 * В предыдущем примере с отскакивающим окном мы могли использовать функцию
 * getArgs() для извлечения из URL дополнительных параметров анимации
 */
var args = getArgs();                    // Получаем аргументы
if (args.x) x = parseInt(args.x);        // Если аргументы определены...
if (args.y) y = parseInt(args.y);        // перекрываем значения по умолчанию
if (args.w) w = parseInt(args.w);
if (args.h) h = parseInt(args.h);
if (args.dx) dx = parseInt(args.dx);
if (args.dy) dy = parseInt(args.dy);
if (args.interval) interval = parseInt(args.interval);
```

Кроме того, что объект Location обладает этими свойствами, он может использоваться так, как если бы сам был элементарным строковым значением. Прочитав значение объекта Location, мы получим ту же строку, которую получили бы при чтении свойства href объекта (поскольку объект Location имеет подходящий метод toString()). Однако, что значительно более инте-

ресно, можно присвоить новую строку URL свойству location окна. Такое присваивание URL объекту Location имеет важный побочный эффект: оно приводит к загрузке и отображению в броузере содержимого присвоенного вами URL. Вот так, например, можно присвоить URL свойству location:

```
// Если у вас старый броузер, который не отображает
// DHTML-содержимое, переходим на страницу, содержащую статический HTML
if (parseInt(navigator.appVersion) < 4)
    location = "staticpage.html";
```

Нетрудно себе представить, что возможность заставить броузер загружать в окна указанные веб-страницы – это очень важная технология программирования. И хотя можно было бы ожидать, что существует метод, предназначенный для отображения в броузере новой веб-страницы, для достижения этого результата поддерживается именно технология присваивания URL свойству location окна. Код для установки значения этого свойства приведен далее в этой главе в примере 13.6.

Несмотря на то что объект Location не имеет метода, выполняющего ту же функцию, что и присваивание URL непосредственно свойству location окна, этот объект поддерживает два метода (добавленные в JavaScript 1.1). Метод reload() заново загружает текущую отображаемую страницу с веб-сервера. Метод replace() загружает и отображает указанный URL. Но вызов этого метода для данного URL отличается от присваивания этого URL свойству location окна. Когда вызывается replace(), указанный URL заменяет текущий URL в списке истории просмотра, а не создает новую запись в списке. Следовательно, если для переписывания одного документа другим вызывается метод replace(), кнопка Back не возвращает пользователя обратно к исходному документу, как это происходит при загрузке нового документа путем присваивания URL свойству location. Для сайтов, использующих фреймы и отображающих много временных страниц (возможно сгенерированных CGI-сценарием), применение replace() часто оказывается полезным. Поскольку временные страницы не хранятся в списке истории, от кнопки Back пользователь может добиться больше толка.

И наконец, не путайте свойство location объекта Window, ссылающееся на объект Location, со свойством location объекта Document, которое просто представляет собой доступную только для чтения строку без каких-либо особенностей, присущих объекту Location. Свойство document.location – это синоним document.URL, которое в JavaScript 1.1 является более предпочтительным именем для этого свойства (так как позволяет избежать потенциальной путаницы). В большинстве случаев document.location совпадает с location.href. Однако когда происходит перенаправление на стороне сервера, document.location содержит URL, который был загружен, а location.href – запрошенный изначально URL.

13.10. Объект History

Свойство `history` объекта Window ссылается на объект History данного окна. Объект History был изначально разработан для моделирования истории просмотра страниц в окне в виде массива недавно открытых URL. Однако этот замысел оказался неудачным, – по серьезным причинам, относящимся к безопасности и секретности, сценарию почти никогда нельзя дать доступ к списку веб-сайтов, ранее посещенных пользователем. Поэтому элементы массива объекта History практически никогда реально не доступны для сценариев (кроме случаев, когда пользователь предоставил на это права в Netscape 4 и более поздних версиях подписанному сценарию). Свойство `length` объекта History доступно, но не предоставляет какой-либо полезной информации.

Рис. 13.3. *Навигационная панель*

Хотя элементы массива недоступны, объект History поддерживает три метода (которые могут использоваться обычными, неподписанными сценариями во всех версиях броузеров). Методы `back()` и `forward()` перемещаются вперед и назад по истории просмотра данного окна (или фрейма), заменяя текущий отображаемый документ на ранее просмотренный. Аналогичные события происходят, когда пользователь щелкает в броузере по кнопкам Back и Forward. Третий метод, `go()`, принимает целый аргумент и пропускает несколько

страниц вперед или назад в списке истории. К сожалению, выполнение `go()` в Netscape 2 и 3 сопровождается ошибками, он характеризуется несовместимым поведением в Internet Explorer 3, и применения этого метода в броузерах ранее четвертого поколения следует избегать.

Пример 13.6 иллюстрирует работу с методами `back()` и `forward()` объекта History, а также добавление навигационной панели к веб-сайту с фреймами с помощью объекта Location. Вид панели навигации представлен на рис. 13.3. Обратите внимание, что в этом примере код JavaScript оперирует несколькими фреймами (что мы вскоре обсудим). Он также содержит простую HTML-форму и использует JavaScript для чтения и записи значений формы. Поведение форм подробно рассматривается в главе 15.

Пример 13.6. Панель навигации, использующая объекты History и Location

```
<!--  Этот файл реализует навигационную панель, предназначенную для фрейма
      в нижней части окна. Включите ее в набор фреймов следующим образом:
        <frameset rows="*,75">
        <frame src="about:blank">
        <frame src="navigation.html">
        </frameset>
-->

<script>
// Эта функция вызывается кнопкой Back на нашей навигационной панели
function go_back()
{
    // Сначала очистим поле ввода URL нашей формы
    document.navbar.url.value = "";

    // Затем используем объект History главного фрейма для возврата назад
    parent.frames[0].history.back();

    // Ждем одну секунду и затем обновляем поле ввода URL в форме из
    // свойства location.href главного фрейма. Ожидание может потребоваться
    // для того, чтобы произошла синхронизация со свойством location.href.
    setTimeout("document.navbar.url.value = parent.frames[0].location.href;",
               1000);
}

// Эта функция вызывается кнопкой Forward навигационной панели;
// она работает точно так же, как предыдущая
function go_forward()
{
    document.navbar.url.value = "";
    parent.frames[0].history.forward();
    setTimeout("document.navbar.url.value = parent.frames[0].location.href;",
               1000);
}

// Эта функция вызывается кнопкой Go навигационной панели, а также
// при подтверждении формы (когда пользователь нажимает Return)
```

```
function go_to()
{
    // Просто устанавливаем свойство location главного фрейма равным
    // значению, набранному пользователем
    parent.frames[0].location = document.navbar.url.value;
}
</script>

<!-- Это форма с обработчиками событий, вызывающими определенные выше
  -- функции -->
<form name="navbar" onsubmit="go_to(); return false;">
  <input type="button" value="Back" onclick="go_back();">
  <input type="button" value="Forward" onclick="go_forward();">
  URL:
  <input type="text" name="url" size="50">
  <input type="button" value="Go" onclick="go_to();">
</form>
```

13.11. Работа с несколькими окнами и фреймами

Большинство примеров клиентского JavaScript, которые мы видели до этого, затрагивали только одно окно или фрейм. В реальном мире приложения JavaScript часто работают с несколькими окнами или фреймами. Вспомните, что фреймы в окне представлены объектами Window; JavaScript не делает особого различия между окнами и фреймами. В наиболее интересных приложениях имеется код JavaScript, исполняющийся независимо в нескольких окнах. В следующих разделах объясняется, как код JavaScript в каждом из окон может взаимодействовать и влиять на другие окна и на сценарии, работающие в этих окнах.

13.11.1. Отношения между фреймами

Мы уже видели, что метод `open()` объекта Window возвращает объект Window, представляющий только что созданное окно. Также мы видели, что это новое окно имеет свойство `opener`, ссылающееся на первоначальное окно. Так, два окна могут ссылаться друг на друга, и каждое из них может читать свойства и вызывать методы другого. То же самое возможно для фреймов. Любой фрейм в окне может ссылаться на любой другой фрейм при помощи свойств `frames`, `parent` и `top` объекта Window.

Любое окно имеет свойство `frames[]`. Это свойство ссылается на массив объектов Window, каждый из которых представляет содержащийся внутри окна фрейм. (Если окно не содержит фреймов, массив `frames[]` пуст и значение `frames.length` равно нулю.) Следовательно, окно (или фрейм) может ссылаться на свой первый подфрейм как на `frames[0]`, на второй подфрейм – как на `frames[1]` и т. д. Аналогично, код JavaScript, работающий в окне, может следующим образом ссылаться на третий подфрейм второго фрейма этого окна:

```
frames[1].frames[2]
```

Каждое окно также имеет свойство parent, ссылающееся на объект Window, в котором оно содержится. Следовательно, первый фрейм в окне может ссылаться на смежный с ним фрейм (второй фрейм того же окна) так:

```
parent.frames[1]
```

Если окно является окном верхнего уровня, а не фреймом, parent просто ссылается на само окно:

```
parent == self;  // Для любого окна верхнего уровня
```

Если фрейм находится внутри другого фрейма, содержащегося в окне верхнего уровня, то он может ссылаться на окно верхнего уровня как parent.parent. Однако в качестве универсального сокращения имеется свойство top: независимо от глубины вложенности фрейма его свойство top ссылается на содержащее его окно самого верхнего уровня. Если окно Window представляет окно верхнего уровня, top просто ссылается на само окно. Для фреймов, непосредственно принадлежащих окну верхнего уровня, свойство top совпадает со свойством parent.

Фреймы обычно создаются с помощью тегов <frameset> и <frame>. Однако в HTML 4, реализованном в IE 4 и более поздних версиях и в Netscape 6 и более поздних версиях, может также использоваться тег <iframe>, создающий в документе плавающий фрейм. Для JavaScript фреймы, созданные с помощью <iframe>, – это то же самое, что фреймы, созданные с помощью <frameset> и <frame>. Все обсуждавшееся выше применимо к обоим видам фреймов.

Рисунок 13.4 иллюстрирует эти отношения между фреймами и показывает, как код, работающий в одном фрейме, может ссылаться на любой другой фрейм посредством свойств frames, parent и top. Здесь окно броузера, содержит два фрейма – один над другим. Второй фрейм (больший, расположенный внизу) сам содержит три подфрейма, расположенных бок о бок.

Разобравшись с отношениями между фреймами, вы можете вернуться к примеру 13.6 и обратить в этот раз особое внимание на то, как код (предназначенный для запуска во втором фрейме) ссылается на свойства history и location первого фрейма.

13.11.2. Имена окон и фреймов

Второй, необязательный аргумент обсуждавшегося ранее метода open() – это имя только что созданного окна. Создавая фрейм с помощью тега <frame>, можно с помощью атрибута name задать его имя. Важным основанием для задания имен окон и фреймов является то, что эти имена могут затем использоваться в качестве значений атрибута target тегов <a>, <map> и <form>. Это значение сообщает броузеру, где вы хотите показывать результаты активизации ссылки, щелчка по карте изображения или подтверждения формы.

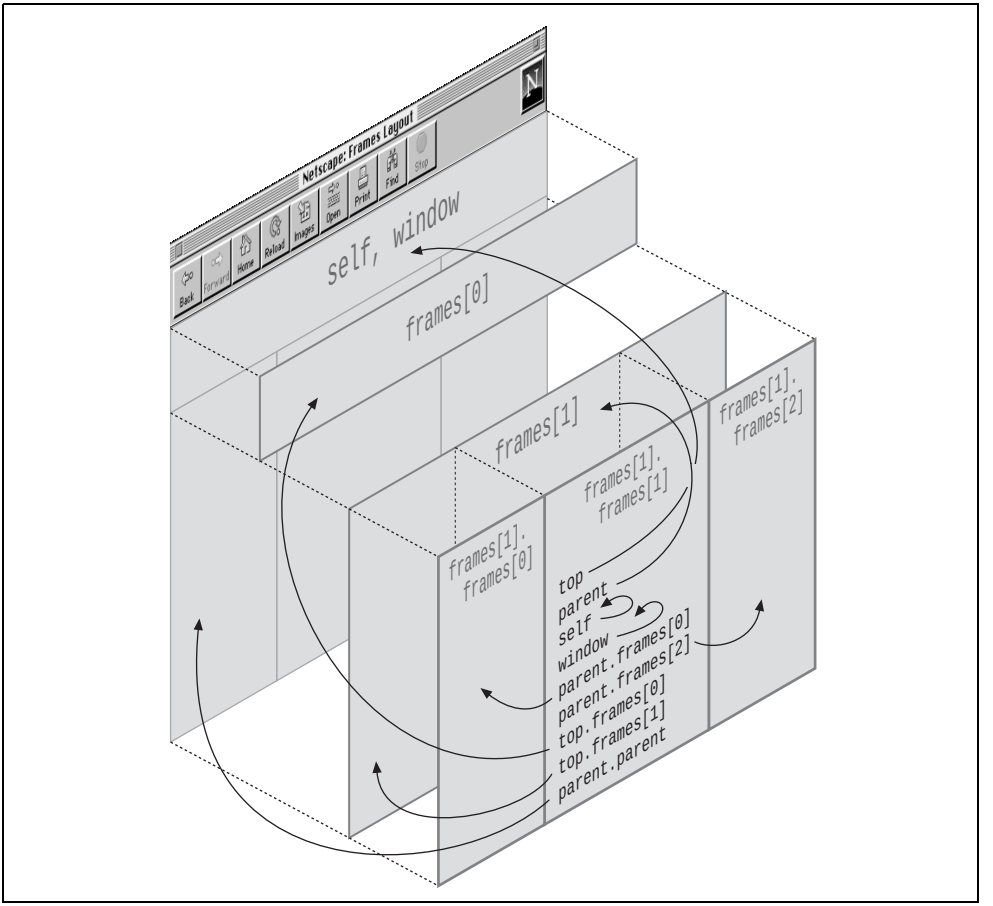

Рис. 13.4. *Отношения между фреймами*

Например, если у вас имеется два окна, одно с именем `table_of_contents`, а другое – `mainwin`, в окне `table_of_contents` может быть следующий HTML-код:

```
<a href="chapter01.html" target="mainwin">
  Глава 1. Введение
</a>
```

Когда пользователь щелкает по этой гиперссылке, броузер загружает указанный URL, но вывод осуществляется не в окно, в котором находится ссылка, а в окно с именем `mainwin`. Если окно с именем `mainwin` отсутствует, щелчок по ссылке создает новое окно с этим именем, и указанный URL загружается в это окно.

Атрибуты `target` и `name` являются частью HTML и работают без вмешательства JavaScript, но есть и связанные с JavaScript причины присваивания имен фреймам. Мы видели, что в любом объекте Window имеется массив `frames[]`,

содержащий ссылки на все фреймы окна (или фрейма) независимо от того, есть у них имена или нет. Однако если фрейму дано имя, ссылка на этот фрейм также сохраняется в новом свойстве родительского объекта Window. Имя нового свойства совпадает с именем фрейма. Следовательно, вы можете создать фрейм с помощью следующего HTML-кода:

```
<frame name="table_of_contents" src="toc.html">
```

Теперь вможно ссылаться на этот фрейм из другого, смежного:

```
parent.table_of_contents
```

Такой код проще читать и понимать, чем если индекс массива жестко закодирован (и вы зависите от него), что неизбежно в случае безымянного фрейма:

```
parent.frames[1]
```

Свойство `name` любого объекта Window содержит имя этого окна. В JavaScript 1.0 это свойство доступно только для чтения. Однако в JavaScript 1.1 и более поздних версиях программист может устанавливать значение этого свойства, изменяя таким образом имя окна или фрейма. Часто эта возможность применяется для установки имени исходного окна броузера. Когда запускается броузер, у исходного окна нет имени, поэтому оно не может выступать в качестве значения атрибута `target`. Но если установить для этого окна свойство `name`, его можно будет помещать в атрибуты `target`.

13.11.3. JavaScript во взаимодействующих окнах

Объект Window выступает в качестве глобального объекта для кода клиентского JavaScript, а окно – в качестве контекста исполнения для всего содержащегося в нем кода JavaScript. Это относится и к фреймам: каждый фрейм представляет собою независимый контекст исполнения JavaScript. Каждый объект Window является отдельным глобальным объектом, поэтому в каждом окне определено свое собственное пространство имен и свой собственный набор глобальных переменных. Если посмотреть с точки зрения работы с несколькими фреймами или окнами, то глобальные переменные уже не кажутся такими глобальными!

Несмотря на то что каждое окно или фрейм определяет независимый контекст исполнения JavaScript, это не значит, что код, работающий в одном окне, изолирован от кода, работающего в других окнах. Код, работающий в одном фрейме, имеет в вершине своей цепочки областей видимости объект Window, отличный от того, который имеет код, работающий в другом фрейме. Однако код из обоих фреймов исполняется одним и тем же интерпретатором JavaScript, в одной и той же среде. Как мы видели, фрейм может ссылаться на любой другой фрейм с помощью свойств `frames`, `parent` и `top`. Поэтому, хотя код JavaScript в разных фреймах исполняется с различными цепочками областей видимости, тем не менее код в одном фрейме может обращаться к переменным и функциям, определенным в коде другого фрейма, и использовать их.

Предположим, что код во фрейме A определяет переменную i:

```
var i = 3;
```

Это переменная представляет собой свойство глобального объекта, то есть свойство объекта Window. Код во фрейме A может явно ссылаться на эту переменную как на свойство с помощью любого из этих двух выражений:

```
window.i
self.i
```

Теперь предположим, что у фрейма A имеется смежный фрейм B, который хочет установить значение переменной i, определенной в коде фрейма A. Если фрейм B просто присвоит значение переменной i, он лишь успешно создаст новое свойство своего собственного объекта Window. Поэтому он должен явно сослаться на свойство i в смежном объекте с помощью следующего кода:

```
parent.frames[0].i = 4;
```

Вспомните, что ключевое слово function, определяющее функцию, объявляет переменную так же, как ключевое слово var. Если код JavaScript во фрейме A объявляет функцию f, эта функция определена только внутри фрейма A. Код во фрейме A может вызывать функцию f следующим образом:

```
f();
```

Однако код во фрейме B должен ссылаться на f как на свойство объекта Window фрейма A:

```
parent.frames[0].f();
```

Если код во фрейме B часто вызывает эту функцию, можно присвоить ее переменной фрейма B, так чтобы было удобнее ссылаться на функцию:

```
var f = parent.frames[0].f;
```

Теперь код во фрейме B может вызывать функцию как f() точно так же, как код фрейма A.

Разделяя подобным образом функции между фреймами или окнами, очень важно помнить о правилах лексического контекста. Функции исполняются в том контексте, в котором они определены, а не в том, из которого они вызываются. Следовательно, продолжая предыдущий пример, если функция f ссылается на глобальные переменные, поиск этих переменных выполняется в свойствах фрейма A, даже когда функция вызывается из фрейма B.

Если не обращать на это особого внимания, могут получаться программы, ведущие себя неожиданным и запутанным образом. Предположим, что вы определили в секции <head> документа, содержащего несколько фреймов, следующую функцию, думая, что она поможет вам при отладке:

```
function debug(msg) {
    alert("Отладочное сообщение от фрейма: " + name + "\n" + msg);
}
```

Код JavaScript в каждом из ваших фреймов может ссылаться на эту функцию как на `top.debug()`. Однако при ее вызове функция будет искать переменную `name` в контексте окна верхнего уровня, в котором определена функция, а не в контексте фрейма, из которого она вызвана. Поэтому отладочные сообщения всегда будут содержать имя окна верхнего уровня, а не имя фрейма, посылающего сообщение, как это предполагалось.

Помните, что конструкторы – это тоже функции, поэтому когда вы определяете класс объектов с функцией-конструктором и связанным с ним объектом-прототипом, этот класс определен только для одного окна. Вспомним класс Complex, который мы определили в главе 8, и рассмотрим следующий документ HTML с несколькими фреймами:

```
<head>
<script src="Complex.js"></script>
</head>
<frameset rows="50%,50%">
  <frame name="frame1" src="frame1.html">
  <frame name="frame2" src="frame2.html">
</frameset>
```

Код JavaScript в файлах *frame1.html* и *frame2.html* не может создавать объект Complex с помощью примерно такого выражения:

```
var c = new Complex(1,2);  // Не работает ни из одного фрейма
```

Он должен явно ссылаться на функцию-конструктор:

```
var c = new top.Complex(3,4);
```

В качестве альтернативы код в любом фрейме может определять собственные переменные для более удобного обращения к функции-конструктору:

```
var Complex = top.Complex;
var c = new Complex(1,2);
```

В отличие от пользовательских конструкторов, предопределенные конструкторы автоматически определены во всех окнах. Заметьте, однако, что каждое окно имеет независимую копию конструктора и независимую копию объекта-прототипа конструктора. Например, каждое окно имеет собственную копию конструктора `String()` и объекта `String.prototype`. Поэтому если вы создадите новый метод для работы со строками JavaScript и сделаете его методом класса String, присвоив его объекту `String.prototype` текущего окна, все строки в этом окне смогут использовать новый метод. Однако новый метод не будет доступен строкам, определенным в других окнах. Обратите внимание, что не имеет значения, в каком из окон содержится ссылка на строку; имеет значение только окно, в котором фактически была создана строка.

13.11.4. Пример: цветные фреймы

В примере 13.7 создается набор из девяти фреймов и продемонстрированы несколько приемов, рассмотренных в этой главе. Сценарий в секции `<head>` набора фреймов определяет функцию JavaScript `setcolor()`. Обработчик события `onload` тега `<frameset>` вызывает ее по одному разу для каждого фрейма.

Функции `setcolor()` в качестве аргумента передается объект Window. Она генерирует случайный цвет и использует его в методе `Document.write()` для создания нового пустого документа с окрашенным фоном. Кроме того, `setcolor()` с помощью метода `setTimeout()` планирует собственный повторный запуск через одну секунду. Этот вызов `setTimeout()` — наиболее интересная часть примера. Обратите особое внимание на то, как в нем используются свойства `parent` и `name` объекта Window.

Пример 13.7. Анимация цветных фреймов

```
<head>
<title>Цветные фреймы</title>
<script>
function setcolor(w) {
    // Генерация случайного цвета
    var r = Math.floor((Math.random() * 256)).toString(16);
    var g = Math.floor((Math.random() * 256)).toString(16);
    var b = Math.floor((Math.random() * 256)).toString(16);
    var colorString = "#" + r + g + b;

    // Установка фона фрейма равным случайному цвету
    w.document.write("<body bgcolor='" + colorString + "'></body>");
    w.document.close();

    // Планирование повторного вызова этого метода через одну секунду.
    // Так как мы вызываем метод setTimeout() фрейма, строка будет
    // исполняться в этом контексте, поэтому вы должны указать "parent."
    // перед свойствами окна верхнего уровня.
    w.setTimeout('parent.setcolor(parent.' + w.name + ')', 1000);

    //  то же самое более простым образом:
    // setTimeout('setcolor(' + w.name + ')', 1000);
}
</script>
</head>
<frameset rows="33%,33%,34%" cols="33%,33%,34%"
  onload="for(var i = 0; i < 9; i++) setcolor(frames[i]);">
<frame name="f1" src="javascript:''"><frame name="f2" src="javascript:''">
<frame name="f3" src="javascript:''"><frame name="f4" src="javascript:''">
<frame name="f5" src="javascript:''"><frame name="f6" src="javascript:''">
<frame name="f7" src="javascript:''"><frame name="f8" src="javascript:''">
<frame name="f9" src="javascript:''">
</frameset>
```

14

Объект Document

У любого объекта Window есть свойство `document`. Это свойство ссылается на объект Document, представляющий HTML-документ, отображаемый в окне. Объект Document, пожалуй, чаще других используется в клиентском JavaScript. Мы уже видели в этой книге несколько примеров применения метода `write()` объекта Document для вставки динамического содержимого в процессе обработки документа. Кроме часто используемого метода `write()` объект Document определяет свойства, предоставляющие информацию о документе в целом: его URL, дату последнего изменения, URL документа, который на него ссылается, цвета, используемые для отображения, и т. д.

Клиентский JavaScript предназначен для того, чтобы превращать статические HTML-документы в интерактивные программы, и именно объект Document дает JavaScript интерактивный доступ к содержимому статических документов. Кроме свойств, предоставляющих информацию о документе в целом, объект Document имеет несколько очень важных свойств, предоставляющих информацию о содержимом документа. Например, массив `forms[]` содержит объекты Form, представляющие все HTML-формы документа. Массивы `images[]` и `applets[]` содержат объекты, представляющие изображения и апплеты в документе. Эти массивы и объекты, содержащиеся в них, открывают богатые возможности для программ клиентского JavaScript, и большая часть этой главы посвящена их описанию.

Здесь рассматриваются базовые возможности объекта Document, реализованные практически во всех поддерживающих JavaScript броузерах. Более новые версии, такие как IE 4 и более поздние, а также Netscape 6 и более поздние, реализуют полную объектную модель документа (DOM), дающую JavaScript полный доступ ко всему содержимому документа и возможность управлять им. Этим расширенным возможностям DOM посвящена глава 17.

14.1. Обзор объекта Document

В этой главе обсуждаются области применения и значение объекта Document, поэтому она начинается с краткой сводки методов и свойств объекта. В следующих разделах также разъясняется другой материал, важный для понимания оставшейся части главы.

14.1.1. Методы объекта Document

Объект Document определяет четыре ключевых метода. Один из них – метод `write()`, уже нам знакомый, и три других связанных с ним метода:

`close()`

Закрывает или завершает документ, начатый с помощью `open()`.

`open()`

Открывает новый документ, удаляя все содержимое существующего.

`write()`

Дописывает текст к открытому в данный момент документу.

`writeln()`

Выводит текст в открытый в данный момент документ и дополняет его символом перевода строки.

14.1.2. Свойства объекта Document

Объект Document определяет следующие свойства:

`alinkColor`, `linkColor`, `vlinkColor`

Эти свойства описывают цвета гиперссылки. `linkColor` – это цвет неоткрытой ссылки. `vlinkColor` – это цвет ссылки, которая была открыта. `alinkColor` – цвет ссылки во время ее активизации (то есть во время щелчка по ней). Эти свойства соответствуют атрибутам `alink`, `link` и `vlink` тега `<body>`.

`anchors[]`

Массив объектов Anchor, представляющих якоря в документе.

`applets[]`

Массив объектов Applet, представляющих апплеты Java в документе.

`bgColor`, `fgColor`

Цвета фона и переднего плана (то есть текста) в документе. Эти свойства соответствуют атрибутам `bgcolor` и `text` тега `<body>`.

`cookie`

Специальное свойство, позволяющее программам JavaScript читать и писать файлы cookie. Подробности см. в главе 16.

`domain`

Свойство, которое позволяет доверяющим друг другу веб-серверам, принадлежащим одному домену, ослаблять ограничения безопасности на взаимодействие между их веб-страницами. См. главу 21.

`forms[]`

Массив объектов Form, представляющих элементы `<form>` в документе.

`images[]`

Массив объектов Image, представляющих элементы `<img>` в документе.

`lastModified`

Строка, содержащая дату изменения документа.

`links[]`

Массив объектов Link, представляющих гиперссылки документа.

`location`

Устаревший синоним свойства `URL`.

`referrer`

URL документа, если такой существует, содержащего ссылку, приведшую броузер к текущему документу.

`title`

Текст между тегами `<title>` и `</title>` данного документа.

`URL`

Строка, задающая URL, с которого был загружен документ. Значение этого свойства совпадает со значением свойства `location.href` объекта Window, за исключением случая перенаправления на стороне сервера.

14.1.3. Объект Document и стандарты

Объект Document и предоставляемый им для программ JavaScript набор элементов (формы, изображения и ссылки) образуют объектную модель документа. Исторически различные разработчики броузеров реализовывали различные объектные модели, что осложняло создание переносимых программ на JavaScript, использующих развитые средства объектных моделей различных разработчиков. К счастью, World Wide Web Consortium (W3C, см. *http://www.w3.org*) стандартизовал объектную модель документа (DOM) и выпустил две версии этого стандарта, известные как Level 1 и Level 2. Последние версии броузеров, такие как Netscape 6 и IE 5 (и более поздние), реализуют подмножество этих стандартов. Более подробно этот вопрос обсуждается в главе 17.

Модель DOM, описанная в этой главе, предшествует стандартам W3C. Однако благодаря ее повсеместному распространению она является стандартом де-факто и часто называется DOM Level 0. Приемы, описанные в этой главе,

действуют в любом веб-броузере, поддерживающем JavaScript, за исключением очень старых, например Netscape 2. Кроме того, методы и свойства объекта Document, перечисленные выше, были формализованы как часть стандарта DOM Level 1, поэтому они гарантированно будут поддерживаться будущими броузерами.

Важно понимать, что стандарт W3C DOM является объектной моделью документа как для XML-, так и для HTML-документов. В этом стандарте объект Document предоставляет обобщенную функциональность для обоих типов документов. Специфическая для HTML функциональность предоставляется подклассом HTMLDocument. Все свойства и методы объекта Document, описанные в этой главе, специфичны для HTML, и подробные сведения о них можно найти в статье «Document» в части IV «Справочник по клиентскому JavaScript», а также в части V «Справочник по W3C DOM», в статьях «Document» и «HTMLDocument».

14.1.4. Назначение имен объектам документа

Перед тем как начать обсуждение объекта Document и различных предоставляемых им объектов, поговорим о важном общем принципе, о котором будет полезно помнить. Как вы увидите, каждый элемент `<form>` в HTML-документе создает нумерованный элемент в массиве `forms[]` объекта Document. Аналогично каждый элемент `<img>` создает элемент в массиве `images[]`. То же самое относится к тегам `<a>` и `<applet>`, определяющим элементы в массивах `links[]` и `applets[]`.

Кроме того, если в соответствующем теге задан атрибут `name`, то к объектам Form, Image или Applet можно обращаться по имени. Значение этого атрибута можно использовать для доступа к соответствующему объекту как к свойству объекта Document. Предположим, что имеется HTML-документ, содержащий следующую форму:

```
<form name="f1">
<input type="button" value="Щелкни меня">
</form>
```

Если этот тег `<form>` – первый в документе, ваш код JavaScript может применять для ссылки на результирующий объект Form одно из двух выражений:

```
document.forms[0]  // Ссылка на форму по ее позиции в документе
document.f1        // Ссылка на форму по имени
```

На самом деле установка атрибута `name` тега `<form>` также делает объект Form доступным как именованное свойство массива `forms[]`, поэтому на форму также можно ссылаться и двумя следующими выражениями:

```
document.forms.f1      // Синтаксис свойств
document.forms["f1"]   // Синтаксис массивов
```

То же самое относится к изображениям и апплетам: атрибут name в HTML позволяет ссылаться на эти объекты из кода JavaScript по имени.

Часто используемым объектам документа удобнее давать имена, чтобы из сценариев было проще ссылаться на них. В последующих главах вы неоднократно встретитесь с этим приемом.

14.1.5. Объекты документа и обработчики событий

Интерактивный HTML-документ и находящиеся в нем элементы должны реагировать на пользовательские события. Мы кратко обсуждали события и их обработчики в главе 12 и видели несколько примеров простых обработчиков. В этой главе мы увидим намного больше примеров обработчиков событий, так как они играют ключевую роль в работе с объектами документа.

К сожалению, мы должны отложить полноценное обсуждение событий и обработчиков событий до главы 19. А сейчас вспомните, что обработчики событий определяются атрибутами HTML-элементов, такими как onclick и onmouseover. Значениями этих атрибутов должны быть строки кода JavaScript. Этот код исполняется всегда, когда с элементом HTML происходит указанное событие.

Кроме того, есть еще один способ определения обработчиков событий, который будет часто применяться в этой и последующих главах. В этой главе мы увидим, что у объектов документа, таких как Form и Image, есть свойства JavaScript, соответствующие атрибутам тегов HTML <form> и <img>. Например, у тега <img> есть атрибуты src и width, а у объекта Image в JavaScript – соответствующие свойства src и width. То же самое относится к обработчикам событий. Например, тег HTML <a> поддерживает обработчик события onclick, а у представляющего гиперссылку объекта Link в JavaScript есть соответствующее свойство onclick. В качестве другого примера рассмотрим атрибут onsubmit элемента <form>. В JavaScript у объекта Form есть соответствующее свойство onsubmit. Помните, что HTML нечувствителен к регистру, и атрибуты могут быть записаны в нижнем, верхнем или смешанном регистре. В JavaScript имена всех свойств обработчиков событий должны быть записаны в нижнем регистре.

В HTML обработчики событий определяются путем присваивания строки, содержащей код на JavaScript, атрибуту-обработчику события. В JavaScript они определяются путем присваивания функции свойству-обработчику события. Рассмотрим следующий тег <form> и его обработчик события onsubmit:

```
<form name="myform" onsubmit="return validateform();">...</form>
```

В JavaScript вместо использования строки JavaScript-кода, вызывающей функцию и возвращающей ее результат, можно непосредственно присвоить функцию свойству-обработчику события:

```
document.myform.onsubmit = validateform;
```

Обратите внимание, что после имени функции не указаны скобки. Дело в том, что здесь мы не хотим вызывать функцию, а просто присваиваем ссылку на нее. В качестве другого примера рассмотрим следующий тег <a> и его обработчик события onmouseover:

```
<a href="help.html" onmouseover="status='Получите помощь!';">Помощь</a>
```

Если нам известно, что этот тег <a> – первый в документе, мы можем сослаться на соответствующий объект Link как на document.links[0] и установить обработчик события следующим образом:

```
document.links[0].onmouseover = function() { status = ' Получите помощь!'; }
```

Полное описание такого способа назначения обработчиков событий приводится в главе 19.

14.2. Динамически генерируемые документы

Одно из наиболее важных средств объекта Document (и, пожалуй, клиентского JavaScript вообще) – метод write(), позволяющий динамически генерировать содержимое веб-страницы из программ на JavaScript. Этот метод может применяться двумя способами. Первый и самый простой – внутри сценария – для выдачи динамически генерируемого HTML в анализируемый в данный момент документ, как обсуждалось в главе 12. Рассмотрим следующий код, где метод write() осуществляет запись текущей даты и даты последнего изменения документа в изначально статичный документ HTML:

```
<script>
var today = new Date();
document.write("<p>Обращение к документу: " + today.toString());
document.write("<br>Модификация документа: " + document.lastModified);
</script>
```

Такое применение метода write() – очень распространенный прием программирования в JavaScript, и он встретится вам во многих сценариях.

Однако имейте в виду, что посредством метода write() можно выводить код HTML в текущий документ только во время анализа документа, то есть document.write() можно вызывать из тегов <script>, только когда эти сценарии вызываются как часть процесса анализа документа. В частности, если вызвать document.write() из обработчика события, который вызывается после анализа документа, то в результате текст к документу не будет добавлен, а документ перепишется целиком (в том числе и его обработчики событий). Причина этого станет ясна после изучения второго способа применения метода write().

Кроме добавления динамического содержимого в текущий документ в процессе его анализа, write() может использоваться в сочетании с методами open() и close() объекта Document для создания совершенно новых документов внутри окна или фрейма. С пользой применить запись в текущий доку-

мент из обработчика события невозможно, но нет причин, мешающих записывать данные в документ, находящийся в другом окне или фрейме; и этот прием может пригодиться в веб-сайтах, использующих несколько окон или фреймов. Например, код JavaScript, находящийся в одном фрейме, может с помощью следующего кода отображать сообщение в другом фрейме:

```
<script>
// Начинаем создание нового документа, стирая любое содержимое,
// уже находящееся во frames[0]
parent.frames[0].document.open();
// Добавляем содержимое в документ
parent.frames[0].document.write("<hr>Привет от смежного фрейма!<hr>");
// И закрываем созданный нами документ
parent.frames[0].document.close();
</script>
```

Для создания нового документа мы сначала вызываем метод `open()` объекта Document, затем для вывода содержимого документа требуемое количество раз вызываем метод `write()` и, наконец, вызываем метод `close()` объекта Document, указывая, что мы закончили создание документа. Последний шаг важен, – если документ не закрыть, броузер не прекратит индикацию процесса его загрузки. Кроме того, броузер может буферизовать создаваемый нами код HTML, и буферизованные данные могут не выводиться, пока документ не закрыт явно вызовом `close()`.

В отличие от `close()`, вызов `open()` не обязателен. Если вызвать метод `write()` для документа, который уже был закрыт, JavaScript неявно откроет новый HTML-документ, как это произошло бы при вызове `open()`. Это объясняет результат вызова `document.write()` из обработчика событий в том же документе. JavaScript открывает новый документ, при этом текущий документ (и его содержимое, в том числе сценарии и обработчики событий) удаляется. Это никогда не требуется и, кроме того, даже может привести к сбою в некоторых старых броузерах (таких, как Netscape 2). Примите за обязательное правило, что документ никогда не должен вызывать метод `write()` в отношении самого себя из обработчика события.

И еще пара замечаний о методе `write()`. Во-первых, многие не понимают, что метод `write()` может принимать более одного аргумента. Несколько аргументов выдаются один за другим, как если бы с ними была выполнена конкатенация. Поэтому вместо

```
document.write("Привет, "  + username + " Добро пожаловать на мою домашнюю страницу!");
```

можно написать:

```
var greeting = " Привет, ";
var welcome = " Добро пожаловать на мою домашнюю страницу!";
document.write(greeting, username, welcome);
```

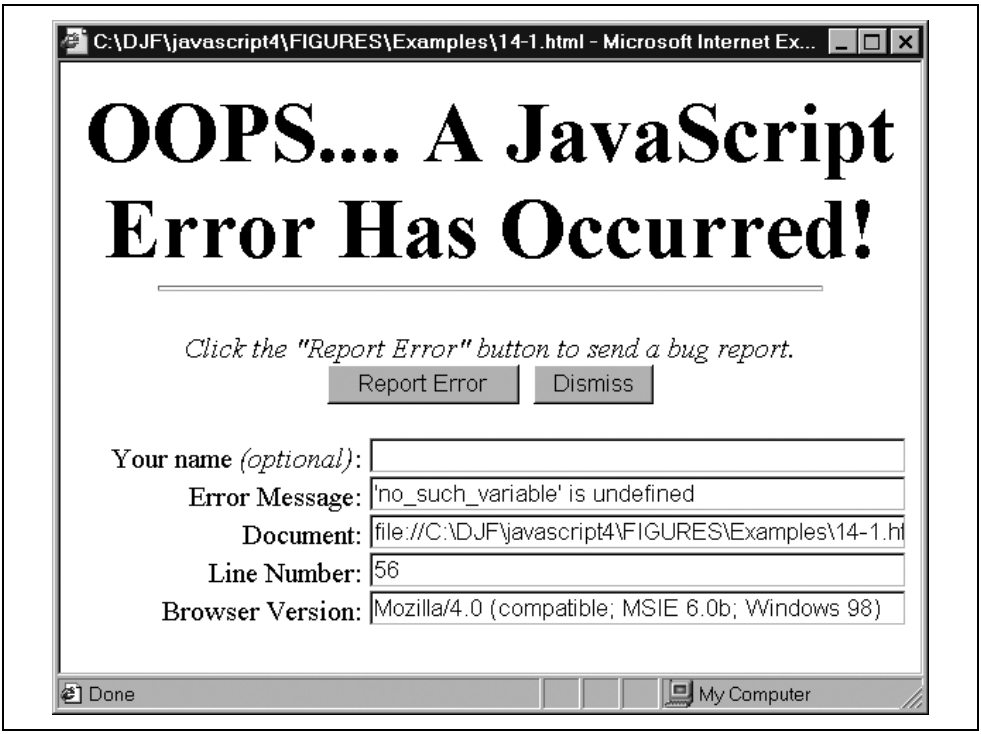

Рис.14.1. *Окно броузера как диалоговое окно*

Во-вторых, объект Document также поддерживает метод `writeln()`, во всем идентичный методу `write()` за исключением того, что добавляет после вывода своих аргументов перевод строки. HTML игнорирует переводы строк, поэтому их наличие обычно не имеет значения, но, как мы скоро увидим, метод `writeln()` может оказаться удобным при работе с не HTML-документами.

В примере 14.1 показано, как с помощью метода `open()` объекта Window и методов объекта Document создавать сложные диалоговые окна. Этот пример регистрирует для окна обработчик события `onerror`, вызываемый при возникновении ошибки JavaScript. Этот обработчик события создает новое окно и с помощью методов объекта Document создает в окне формы HTML. Форма отображает сведения об ошибке и обеспечивает пользователю возможность отправить по электронной почте отчет об ошибке автору кода JavaScript.

Окно на рис. 14.1 создается кодом этого примера. Вспомните из обсуждения обработчика ошибок `onerror` в главе 13, что Netscape 6 не передает правильных параметров функции-обработчику события. И поэтому результат работы данного примера в Netscape 6 не соответствует иллюстрации.

Пример 14.1. Динамическое создание диалогового окна

```
<script>
// Переменная, которая используется для обеспечения уникальности
```

```
// каждого окна сообщений об ошибках
var error_count = 0;

// Установите эту переменную равной вашему адресу email.
var email = "myname@mydomain.com";

// Определяем обработчик ошибок. Он генерирует HTML-форму, чтобы
// пользователь мог сообщить автору об ошибке.

function report_error(msg, url, line)
{
    var w = window.open("",                      // URL (не указан)
                        "error"+error_count++,   // Имя (должно быть уникально)
                        "resizable,status,width=625,height=400");  // Опции
    // Получаем объект Document для нового окна
    var d = w.document;
    // Выводим HTML-документ, включая форму, в новое окно.
    // Обратите внимание, что мы опускаем необязательный вызов document.open()
    d.write('<div align="center">');
    d.write('<font size="7" face="helvetica"><b>');
    d.write('OOPS.... A JavaScript Error Has Occurred!');
    d.write('</b></font><br><hr size="4" width="80%">');
    d.write('<form action="mailto:' + email + '" method=post');
    d.write(' enctype="text/plain">');
    d.write('<font size="3">');
    d.write('<i>Click the "Report Error" button to send a bug report.</i><br>');
    d.write('<input type="submit" value="Report Error">  ');
    d.write('<input type="button" value="Cancel" onclick="self.close();">');
    d.write('</div><div align="right">');
    d.write('<br>Your name <i>(optional)</i>: ');
    d.write('<input size="42" name="name" value="">');
    d.write('<br>Error Message: ');
    d.write('<input size="42" name="message" value="' + msg + '">');
    d.write('<br>Document: <input size="42" name="url" value="' + url + '">');
    d.write('<br>Line Number: <input size="42" name="line" value="'+line +'">');
    d.write('<br>Browser Version: ');
    d.write('<input size="42" name="version" value="'+navigator.userAgent+'">');
    d.write('</div></font>');
    d.write('</form>');
    // Не забудьте закрыть документ по завершении вывода
    d.close();

    // Возвращает true из обработчика ошибок, чтобы JavaScript не выдавал
    // своего диалогового окна с ошибкой
    return true;
}

// Перед тем как обработчик ошибок вступит в силу, мы должны зарегистрировать
// его для определенного окна
self.onerror = report_error;
</script>

<script>
// Следующая строка кода намеренно вызывает ошибку для проверки обработчика
```

```
alert(no_such_variable);
</script>
```

14.2.1. Документы, не содержащие HTML

Вызывая метод `open()` объекта Document без аргументов, мы открываем новый документ HTML. Однако помните, что веб-броузеры могут отображать несколько других форматов данных кроме HTML. Для того чтобы динамически создать и показать документ с другим форматом данных, следует вызвать метод `open()` с одним аргументом, задающим соответствующий тип MIME.[1]

Тип MIME для HTML – это `text/html`. Самый распространенный формат кроме HTML – это обычный текст с типом MIME `text/plain`. Метод `write()` позволяет выводить текст, форматирование которого выполнено с помощью переводов строк, пробелов и символов табуляции, для чего необходимо открыть документ, передав методу `open()` строку «text/plain». Пример 14.2 показывает один из способов, позволяющих это сделать. Он реализует функцию `debug()`, которую можно использовать для вывода отладочных текстовых сообщений из сценариев в появляющееся при необходимости отдельное окно (рис. 14.2).

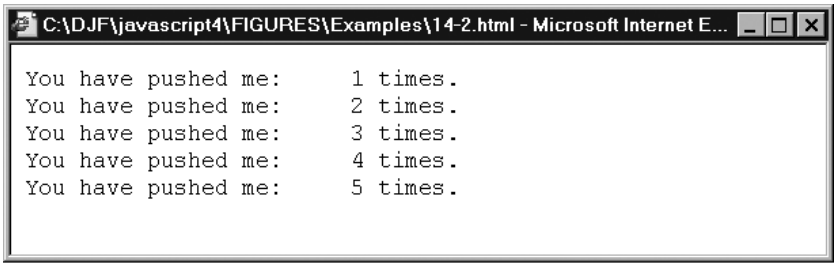

Рис. 14.2. *Окно для текстового отладочного вывода*

Пример 14.2. Создание текстового документа

```
<script>
var _console = null;

function debug(msg)
{
    // Открываем окно при первом вызове функции или после того, как
    // консольное окно было закрыто
    if ((_console == null) || (_console.closed)) {
    _console = window.open("","console","width=600,height=300,resizable");
    // Открываем в окне документ для отображения текста
    _console.document.open("text/plain");
}
```

[1] Этот аргумент метода `open()` не был стандартизован в W3C DOM. Он поддерживается в IE 4 и более поздних версиях, а также в Netscape 3 и 4. Как ни странно, это не работает в Netscape 6, который поддерживает только документы HTML.

```
    _console.focus();                    // Делаем окно видимым
    _console.document.writeln(msg);      // Выводим в него сообщение
    // Обратите внимание, что мы намеренно не вызываем close().
    // Оставив документ открытым, мы получаем возможность позднее
    // дописать в него данные.
}
</script>

<!-- Вот пример использования этого сценария -->
<script>var n = 0;</script>
<form>
<input type="button" value="Push Me"
       onclick="debug('You have pushed me:\t' + ++n + ' times.');">
</form>
```

14.3. Свойства цвета документа

Свойства `bgColor`, `fgColor`, `linkColor`, `alinkColor` и `vlinkColor` объекта Document задают цвет переднего плана, фона и ссылок в документе. Эти свойства доступны для чтения и записи, но они могут устанавливаться только до анализа тега `<body>`. Их можно устанавливать динамически в коде JavaScript, расположенном в секции `<head>` документа, или статически, как атрибуты тега `<body>`, но нельзя установить в других местах. Исключением из этого правила является свойство `bgColor`. Во многих броузерах это свойство можно установить в любой момент, и это приведет к изменению цвета фона окна броузера.[1] Остальные цветовые свойства объекта Document лишь обеспечивают доступ к атрибутам тега `<body>` и не представляют особого интереса.

Все цветовые свойства имеют строковые значения. Цвет можно установить, либо указав одно из предопределенных в HTML имен цветов, либо задав цвет в виде строки из шести шестнадцатеричных цифр в формате *#RRGGBB*, определяющей значения красной, зеленой и синей компонент цвета. Вспомните, что в примере 13.7 атрибуту `bgcolor` тега `<body>` присваивалась строка именно такого формата.

В стандарте W3C DOM цветовые свойства объекта Document признаны устаревшими, и рекомендуется использовать свойства объекта Element, представляющего тег `<body>`. Кроме того, стандарт HTML 4 признает устаревшими цветовые атрибуты тега `<body>` и рекомендует применять каскадные таблицы стилей. Вряд ли разумно писать сценарии, всерьез полагающиеся на цветовые свойства, дважды признанные устаревшими!

---

[1] В Netscape 3 на платформах Unix имеется ошибка, заключающаяся в том, что изменение цвета фона может привести к исчезновению содержимого страницы (обычно до прокрутки или другой перерисовки). В Netscape 6 можно установить `bgColor` только один раз; любые последующие установки игнорируются.

14.4. Информационные свойства документа

Несколько свойств объекта Document предоставляют информацию о документе в целом. Например, следующий код показывает, как с помощью свойств lastModified, title и URL автоматически включить в документ временную метку, позволяющую судить об актуальности (или неактуальности) документа, что может быть полезным при печати последнего.

```
<hr><font size="1">
Document: <i><script>document.write(document.title);</script></i><br>
URL: <i><script>document.write(document.URL);</script></i><br>
Last Update: <i><script>document.write(document.lastModified);</script></i>
</font>
```

Еще одно интересное свойство – referrer. Оно содержит URL, из которого пользователь пришел по ссылке на текущий документ. Одно из возможных применений этого значения состоит в том, чтобы сохранить его в скрытом поле формы на веб-странице. Когда пользователь передает данные формы, можно сохранить информацию из referrer на сервере, чтобы проанализировать ссылки на вашу веб-страницу и выяснить процент посещений, пришедших через разные ссылки. Это свойство также делает возможным трюк, позволяющий предотвратить корректную работу несанкционированных ссылок на вашу страницу. Предположим, что вы хотите разрешить другим сайтам ссылаться только на страницы верхнего уровня, расположенные на вашем сайте. Свойство referrer в сочетании со свойством location объекта Window позволяет перенаправлять любые ссылки из-за пределов вашего сайта на страницу верхнего уровня:

```
<script>
// Если пользователь пришел по внешней ссылке, переходим к домашней странице
if (document.referrer == "" || document.referrer.indexOf("mysite.com") == -1)
    window.location = "http://home.mysite.com";
</script>
```

Конечно, не следует рассматривать этот трюк как серьезную меру безопасности. Важный его недостаток в том, что он не срабатывает в броузерах, не поддерживающих JavaScript, или если пользователь отключил JavaScript.

14.5. Формы

Массив forms[] объекта Document содержит объект Form, представляющий любые элементы <form> в документе. Формы HTML содержат кнопки, текстовые поля ввода и другие элементы ввода, обычно составляющие графический интерфейс веб-приложения, поэтому объект Form очень важен для клиентского JavaScript. В объекте Form имеется свойство elements[], содержащее объекты, представляющие расположенные на форме HTML элементы ввода. Эти объекты Element позволяют программам на JavaScript устанавливать в форме значения по умолчанию и читать введенные пользователем

данные. Они также важны как место, в котором располагаются обработчики событий, добавляющие в программу интерактивность.

Формы и их элементы – настолько большая и важная часть программирования клиентского JavaScript, что заслуживают отдельной главы. Мы вернемся к массиву forms[] и объекту Form в главе 15.

14.6. Изображения

Свойство images[] объекта Document – это массив элементов Image, каждый из которых представляет одно из содержащихся в документе изображений, созданных с помощью тега <img>. Массив images[] и объект Image были добавлены в JavaScript 1.1. Хотя веб-броузеры всегда могли показывать изображения с помощью тега <img>, добавление объекта Image – это большой шаг вперед. Он позволяет программам динамически манипулировать изображениями.

14.6.1. Замена изображения с помощью свойства src

Главная особенность объекта Image заключается в том, что его свойство src доступно как для чтения, так и для записи. Это свойство можно прочитать, чтобы получить URL, с которого было загружено изображение, и, что более важно, его можно установить, заставив броузер загрузить и показать в том же месте новое изображение. Для того чтобы замена была успешной, новое изображение должно иметь те же ширину и высоту, что и первоначальное.

На практике этот прием чаще всего реализуется в ролловерах и осуществляется при перемещении курсора мыши над изображением. Если изображение помещается в гиперссылку, то ролловер представляет собою мощный способ пробудить в пользователе желание щелкнуть по изображению. Вот простой фрагмент HTML, выводящий изображение внутри тега <a> и использующий код JavaScript в обработчиках событий onmouseover и onmouseout для создания ролловера (переключающихся изображений):

```
<a href="help.html"
   onmouseover="document.helpimage.src='images/help_rollover.gif';"
   onmouseout="document.helpimage.src='images/help.gif';">
<img name="helpimage" src="images/help.gif" width="80" height="20" border="0">
</a>
```

Обратите внимание, что в этом фрагменте кода мы указываем в теге <img> атрибут name, чтобы упростить ссылку на соответствующий объект Image из обработчиков событий в теге <a>. Мы использовали атрибут border, чтобы броузер не показывал синюю границу гиперссылки вокруг изображения. Всю работу выполняют обработчики событий тега <a>: они изменяют показываемое изображение, просто устанавливая свойство src изображения равным URL требуемых изображений.

Возможность динамической замены в статическом HTML-документе одного изображения на другое открывает простор для самых разных спецэффектов,

от анимации до цифровых часов, обновляющихся в реальном времени. Немного поразмыслив, вы наверняка придумаете много других потенциальных применений этой технологии.

14.6.2. Внеэкранные изображения и кэширование

Для того чтобы технология замены изображений была жизнеспособной, анимация и другие спецэффекты должны быстро реагировать на действия пользователя. Это означает, что нам требуется какой-либо способ, обеспечивающий предварительную загрузку необходимых изображений в кэш броузера. Чтобы изображение кэшировалось, сначала создадим внеэкранное изображение с помощью конструктора `Image()`. Затем загрузим в него изображение, установив свойство `src` равным желаемому URL (в точности так же, как это делается для изображения на экране). Теперь, когда данный URL используется для экранного изображения, оно будет быстро загружено из кэша броузера (а не через сеть, медленно). Обратите внимание, что мы никогда ничего реально не делаем с созданным нами внеэкранным изображением. В частности, мы *не* присваиваем объект Image внеэкранного изображения элементу массива `images[]` документа.

Фрагмент кода с переключением изображений, рассмотренный в предыдущем разделе, не выполняет предварительной загрузки используемого им переключаемого изображения, поэтому пользователь наверняка заметит задержку в эффекте переключения при первом перемещении указателя мыши над изображением. Чтобы устранить задержку, мы должны изменить код следующим образом:

```
<script>
// Создание внеэкранного изображения и предварительная загрузка изображения
// для переключения. Обратите внимание, что мы не сохраняем ссылку на внеэкранное
// изображение, так как нам больше нечего с ним делать.
(new Image(80,20)).src = "images/help_rollover.gif";
</script>
<a href="help.html"
     onmouseover="document.helpimage.src='images/help_rollover.gif';"
     onmouseout="document.helpimage.src='images/help.gif';">
<img name="helpimage" src="images/help.gif" width="80" height="20" border="0">
</a>
```

Код в примере 14.3 выполняет простую анимацию путем замены изображений и организует предварительную загрузку кадров анимации при помощи внеэкранных изображений. Обратите внимание, что в этом примере мы сохраняем созданные нами внеэкранные изображения, так как они предоставляют удобный способ хранения URL изображений, составляющих анимацию. Для выполнения анимации мы присваиваем свойство `src` одного из внеэкранных изображений свойству `src` расположенного на экране изображения, которое используется для вывода анимации.

Пример 14.3. Анимация, основанная на замене изображений

```
<!-- Изображение, которое будет анимироваться. -->
<!-- Для удобства присваиваем ему имя.         -->
<img name="animation" src="images/0.gif">

<script>
// Создаем группу внеэкранных изображений и выполняем предварительную
// загрузку "кадров" анимации, так что они будут сохранены в кэше до
// того момента, как они нам потребуются.
var aniframes = new Array(10);
for(var i = 0; i < 10; i++) {
    aniframes[i] = new Image();                    // Создаем внеэкранное изображение
    aniframes[i].src = "images/" + i + ".gif";      // Сообщаем ему URL для загрузки
}

var frame = 0;          // Счетчик фреймов отслеживает текущий кадр
var timeout_id = null; // Позволяет нам остановить анимацию с помощью clearTimeout()

// Эта функция осуществляет анимацию. Вызываем ее один раз для запуска анимации.
// Обратите внимание, что мы ссылаемся на экранное изображение по его атрибуту
// name.
function animate() {
    document.animation.src = aniframes[frame].src;   // Выводим текущий кадр
    frame = (frame + 1)%10;                          // Обновляем счетчик кадров
    timeout_id = setTimeout("animate()", 250);       // Другой кадр будет показан
                                                     //  позднее
}
</script>

<form>  <!-- Эта форма содержит кнопки для управления анимацией -->
  <input type="button" value="Start"
         onclick="if (timeout_id == null) animate();">
  <input type="button" value="Stop"
         onclick="if (timeout_id) clearTimeout(timeout_id); timeout_id=null;">
</form>
```

14.6.3. Обработчики событий изображения

В примере 14.3 наша анимация не начинается до тех пор, пока пользователь не щелкнет по кнопке Start, что дает много времени для загрузки наших изображений в кэш. Но как быть в более распространенном случае, когда требуется автоматически начинать анимацию сразу после загрузки всех необходимых изображений? Оказывается, что у изображений, как созданных на экране с помощью тега `<img>`, так и внеэкранных, созданных с помощью конструктора `Image()`, может быть обработчик события `onload`, вызываемый сразу после полной загрузки изображения.

Следующий фрагмент кода показывает, как модифицировать пример 14.3, чтобы с помощью обработчика события подсчитывать количество загруженных изображений и автоматически запускать анимации после их загрузки.

Внеэкранные изображения не являются частью документа HTML, поэтому обработчик события не может быть назначен в виде атрибута HTML. Вместо этого мы просто присвоим функцию свойству onload каждого созданного нами объекта Image. Когда изображение загружается, броузер вызывает заданную функцию.

```
var aniframes = new Array(10);  // Служит для внеэкранного хранения кадров.
var num_loaded_images = 0;      // Сколько изображений уже загружено?

// Эта функция используется как обработчик событий. Она подсчитывает, сколько
// изображений было загружено, и, когда загружены все, запускает анимацию.
function countImages() {
    if (++num_loaded_images == aniframes.length) animate();
}

// Создаем внеэкранные изображения и присваиваем им URL.
// Также назначаем каждому изображению обработчик события, чтобы мы могли
// отслеживать количество загруженных изображений. Обратите внимание, что
// мы назначаем обработчик до присваивания URL, так как в противном случае
// изображение может завершить загрузку (например, если оно уже находится в
// кэше) до назначения обработчика, и в этом случае обработчик никогда не будет
// вызван.
for(var i = 0; i < 10; i++) {
    aniframes[i] = new Image();                 // Создает внеэкранное изображение.
    aniframes[i].onload = countImages;          // Назначает обработчик события
    aniframes[i].src = "images/" + i + ".gif";  // Сообщаем URL для загрузки
}
```

Кроме обработчика события onload объект Image поддерживает еще два обработчика. Обработчик события onerror вызывается, когда при загрузке изображения возникает ошибка, например, когда указанный URL ссылается на неправильные данные. Обработчик onabort вызывается, если пользователь отменяет загрузку изображения (например, щелкнув по кнопке Stop в броузере) до ее завершения. Для любого изображения вызывается один (и только один) из этих обработчиков.

Кроме обработчиков событий у каждого объекта Image есть свойство complete. Это свойство устанавливается в false в процессе загрузки изображения и становится равным true, как только изображение будет загружено или как только броузер прекратит попытки его загрузить. Другими словами, свойство complete становится равным true после того, как будет вызван один из трех возможных обработчиков событий.

14.6.4. Другие свойства изображения

У объекта Image есть еще несколько свойств. Большинство из них просто отражают атрибуты создавшего изображение тега <img>. Свойства width, height, border, hspace и vspace принимают целочисленные значения и задают размер изображения, ширину рамки и размер горизонтальных и вертикальных полей. Данные свойства указываются в атрибутах тега <img>, имеющих те же

имена. В Netscape 3 и 4 эти свойства доступны только для чтения, но в IE 4 и Netscape 6 (и в более поздних версиях) значения этих свойств можно также устанавливать для динамического изменения размеров, рамки или полей изображения.

Свойство `lowsrc` объекта Image соответствует атрибуту `lowsrc` тега `<img>` и задает URL дополнительного изображения, показываемого, если страница просматривается на устройстве с низким разрешением. Свойство `lowsrc` – это строка, доступная для чтения и для записи, как и `src`, но в отличие от `src`, установка `lowsrc` не приводит к тому, что броузер загружает и показывает новое изображение с низким разрешением. Для того чтобы реализовать анимацию или другой спецэффект как для низкого, так и для высокого разрешения, не забудьте обновить свойство `lowsrc` перед установкой свойства `src`. Если броузер работает на устройстве с низким разрешением, во время установки значения `src` он загрузит изображение, заданное свойством `lowsrc`.

14.6.5. Пример замены изображения

Замена изображений – технология весьма многогранная, поэтому обсуждение объекта Image мы завершим расширенным примером. Пример 14.4 определяет класс ToggleButton, в котором замена изображений применяется для графической имитации флажка. Этот класс работает с изображениями, предоставленными программистом, поэтому качество графики может быть более высоким, чем у простых графических элементов, используемых стандартным объектом HTML CheckBox. На рис. 14.3 показано, как эти переключатели выглядят на веб-странице. Это сложный, реальный пример, и его стоит изучить внимательно.

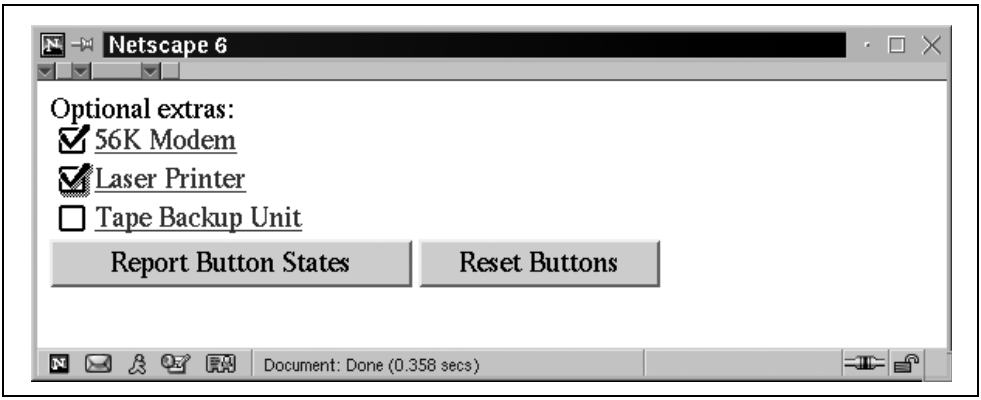

Рис. 14.3. *Флажки, реализованные с помощью замены изображений*

Пример 14.4. Реализация флажков с помощью замены изображений

```
<script language="JavaScript1.1">
// Это функция-конструктор для нашего нового класса ToggleButton. При ее вызове
// создается объект ToggleButton, и в указанный документ в текущую позицию
// выводятся требуемые теги <a> и <img>. Следовательно, ее нельзя вызывать
```

```
// для текущего документа из обработчика событий
// Аргументы:
//    document: Объект Document, в котором должны быть созданы флажки.
//    checked:  Логическое значение, указывающее, выбран ли флажок
//              изначально.
//    label:    Необязательная строка, задающая текст, который выводится
//              после флажка.
//    onclick:  Необязательная функция, вызываемая при щелчке по флажку.
//              Ей передается логическое значение, обозначающее
//              новое состояние флажка. Вы также можете передать строку, которая
//              преобразуется в функцию с логическим аргументом "state".
function ToggleButton(document, checked, label, onclick)
{
    // При первом вызове (и только при первом вызове) функции мы должны
    // выполнить некоторые специальные действия. Во-первых, когда создан
    // объект-прототип, мы можем установить наши методы.
    // Во-вторых, мы должны загрузить используемые нами изображения.
    // Тогда к тому моменту, когда они нам понадобятся,
    // эти изображения будут находиться в кэше.
    if (!ToggleButton.prototype.over) {
        // Инициализация объекта-прототипа для создания наших методов
        ToggleButton.prototype.over = _ToggleButton_over;
        ToggleButton.prototype.out = _ToggleButton_out;
        ToggleButton.prototype.click = _ToggleButton_click;

        // Теперь создаем массив объектов Image и присваиваем им URL. URL
        // изображений настраиваемы и хранятся в свойстве-массиве самой
        // функции-конструктора. Ниже выполняется их инициализация. Из-за
        // ошибки в Netscape нам приходится сохранять ссылки на эти
        // изображения, поэтому мы сохраняем массив в свойстве конструктора,
        // а не в локальной переменной.
        ToggleButton.images = new Array(4);
        for(var i = 0; i < 4; i++) {
            ToggleButton.images[i] = new Image(ToggleButton.width,
                                               ToggleButton.height);
            ToggleButton.images[i].src = ToggleButton.imagenames[i];
        }
    }

    // Сохраняем некоторые из переданных аргументов.
    this.document = document;
    this.checked = checked;

    // Помните, что в данный момент мышь не находится над флажком.
    this.highlighted = false;

    // Сохраняем аргумент onclick, чтобы обработчик мог вызываться
    // при щелчке по флажку.
    // Если это не функция, пытаемся преобразовать ее в функцию
    // с аргументом "state".
    this.onclick = onclick;
```

```
    if (typeof this.onclick == "string")
        this.onclick = new Function("state", this.onclick);

    // Выясняем, в каком элементе массива document.images[]
    // будут храниться изображения для этого флажка.
    var index = document.images.length;

    // Теперь выводим HTML-код флажка. Используем теги <a> и <img>.
    // Обработчики событий, которые мы здесь выводим, могут показаться
    // запутанными, но они критически важны для функционирования этого
    // класса. Свойство"_tb", а также методы over(), out() и click()
    // будут определены ниже.
    document.write(' <a href="about:blank" ' +
      'onmouseover="document.images[' + index + '].._tb.over();return true;" '+
      'onmouseout="document.images[' + index + ']._tb.out()" '+
      'onclick="document.images[' + index + '].._tb.click(); return false;">');
    document.write('<img src="' + ToggleButton.imagenames[this.checked+0] +'"'+
                   ' width=' + ToggleButton.width +
                   ' height=' + ToggleButton.height +
                   ' border="0" hspace="0" vspace="0" align="absmiddle">');
    if (label) document.write(label);
    document.write('</a></br>');

    // Теперь, когда мы создали тег <img>, сохраняем ссылку на созданный
    // объект Image в объекте ToggleButton
    this.image = document.images[index];

    // Также делаем ссылку в обратном направлении, из объекта Image на
    // данный объект ToggleButton. Делаем это путем определения свойства "_tb"
    // объекта Image.
    this.image._tb = this;
}

// Это будет методом over()
function _ToggleButton_over()
{
    // Изменяем изображение и запоминаем состояние
    this.image.src = ToggleButton.imagenames[this.checked + 2];
    this.highlighted = true;
}

// Это будет методом out()
function _ToggleButton_out()
{
    // Изменяем изображение и запоминаем невыделенное состояние
    this.image.src = ToggleButton.imagenames[this.checked + 0];
    this.highlighted = false;
}

// Это будет методом click()
function _ToggleButton_click()
```

```
{
    // Переключаем состояние флажка, изменяем изображение и вызываем
    // метод onclick, если он определен для этой кнопки.
    this.checked = !this.checked;
    this.image.src = ToggleButton.imagenames[this.checked+this.highlighted*2];
    if (this.onclick) this.onclick(this.checked);
}

// Инициализируем статические свойства класса, описывающие изображения флажка.
// Это лишь значения по умолчанию. Программы могут перекрывать их путем
// присвоения новых значений. Но перекрытие должно выполняться *до того, как*
// будет создан хотя бы один флажок ToggleButton.
ToggleButton.imagenames = new Array(4);           // Создаем массив.
ToggleButton.imagenames[0] = "images/button0.gif";// Не отмеченная кнопка
ToggleButton.imagenames[1] = "images/button1.gif";// Кнопка с отметкой
ToggleButton.imagenames[2] = "images/button2.gif";    // Не отмеченная, но выделенная
ToggleButton.imagenames[3] = "images/button3.gif";// Отмеченная и выделенная
ToggleButton.width = ToggleButton.height = 25;    // Размер всех изображений
</script>

<!-- Так может использоваться класс ToggleButton -->
Необязательные дополнения:<br>
<script language="JavaScript1.1">
// Создаем объекты ToggleButton и выводим HTML-код, реализующий их.
// Один из флажков не имеет обработчика события щелчка, другой имеет в качестве
// обработчика функцию, а третий - строку.
var tb1 = new ToggleButton(document, true, "56K Modem");
var tb2 = new ToggleButton(document, false, "Laser Printer",
                           function(clicked) {alert("printer: " + clicked);});
var tb3 = new ToggleButton(document, false, "Tape Backup Unit",
                          "alert('Tape backup: ' + state)");
</script>

<!-- Так объекты ToggleButton можно использовать из обработчиков событий -->
<form>
<input type="button" value="Report Button States"
       onclick="alert(tb1.checked + '\n' + tb2.checked + '\n' + tb3.checked)">
<input type="button" value="Reset Buttons"
       onclick="if (tb1.checked) tb1.click();
                if (tb2.checked) tb2.click();
                if (tb3.checked) tb3.click();">
</form>
```

14.7. Ссылки

Массив `links[]` объекта Document содержит объекты Link, представляющие все гиперссылки в документе. Вспомните, что гиперссылки HTML кодируются с помощью атрибута `href` тега `<a>`. В JavaScript 1.1 и более поздних версиях тег `<area>` в клиентской карте изображений также создает объект Link в массиве `links[]` объекта Document.

Объект Link представляет URL гиперссылки и содержит все свойства, которые содержит объект Location (введенный в главе 13). Например, свойство `href` объекта Link содержит полный текст URL, на который указывает ссылка, а свойство `hostname` – только имя хоста из этого URL. Полный список этих свойств, связанных с URL, приведен в части IV «Справочник по клиентскому JavaScript».

В примере 14.5 показана функция, генерирующая список всех ссылок в документе. Обратите внимание на применение методов `write()` и `close()` объекта Document для динамического создания документа, что обсуждалось ранее в этой главе.

Пример 14.5. Выдача списка ссылок в документе

```
/*
 * ФАЙЛ: listlinks.js
 * Список всех ссылок в указанном документе в новом окне
 */
function listlinks(d) {
    var newwin = window.open("", "linklist",
                             "menubar,scrollbars,resizable,width=600,height=300");

    for (var i = 0; i < d.links.length; i++) {
        newwin.document.write('<a href="' + d.links[i].href + '">')
        newwin.document.write(d.links[i].href);
        newwin.document.writeln("</a><br>");
    }
    newwin.document.close();
}
```

14.7.1. Ссылки, «веб-черви» и безопасность в JavaScript

Одно из очевидных применений объекта Link и массива `links[]` состоит в написании программ, именуемых «веб-червями». Такая программа работает в одном окне или фрейме броузера и читает веб-страницы в другое окно или фрейм (путем установки свойства `location` объекта Window). Для каждой считываемой страницы она просматривает массив `links[]` и рекурсивно обходит их. Будучи аккуратно написанной (то есть не уходя в бесконечную рекурсию и в обход ссылок по кругу), такая программа может, например, использоваться для генерирования списка всех веб-страниц, доступных из данной начальной страницы. Этот список может быть очень полезен при сопровождении сайта.

Однако не надейтесь обойти с помощью таких приемов весь Интернет. По соображениям безопасности JavaScript не позволяет неподписанному сценарию в одном окне или фрейме прочитать свойства (такие как `document.links`) другого окна или фрейма, если только оба окна не показывают документы, взятые с одного веб-сервера. Это ограничение предотвращает серьезные нарушения безопасности. Представьте, что работник большой, беспокоящейся

о безопасности компании просматривает Интернет через корпоративный брандмауэр, а в другом окне броузера просматривает внутреннюю информацию компании из корпоративной сети. Без описанных нами ограничений безопасности сценарий с какого-то произвольного интернет-сайта мог бы подсмотреть, что происходит в другом окне. Авторы подсматривающего сценария, может быть, и не соберут много полезной информации из массива links[] внутренних документов, но это, тем не менее, будет серьезным нарушением безопасности.

Обсуждаемая программа «веб-червь» не угрожает безопасности или секретности в Интернете, но, к сожалению, и она зависит от общих ограничений безопасности JavaScript, не дающих уходить слишком далеко от сайта, с которого был загружен сценарий. (Когда «червь» загружает страницу с другого сайта, он работает так, словно на этой странице нет ссылок.) Полное обсуждение вопросов безопасности в JavaScript, включая способы обхода ограничений безопасности с помощью подписанных сценариев, можно найти в главе 21.

14.7.2. Обработчики событий в ссылках

Объект Link поддерживает несколько интересных обработчиков событий. Мы уже видели обработчик события onmouseover в разделе 13.3 «Строка состояния», где он использовался с тегами <a> и <area> для изменения сообщения в строке состояния броузера, когда указатель мыши перемещается над ссылкой. Обработчик события onclick вызывается, когда пользователь щелкает по гиперссылке. В JavaScript 1.1 и более поздних версиях, если этот обработчик события возвращает false, броузер не открывает ссылку. В JavaScript 1.1 и тег <a> и тег <area> поддерживают обработчик события onmouseout. Он просто является противоположностью обработчику onmouseover и вызывается, когда указатель мыши уходит от гиперссылки.

Модель обработки событий в JavaScript 1.2 стала значительно более общей, и ссылки поддерживают много других обработчиков событий. Более подробная информация приведена в главе 19.

И наконец, стоит упомянуть, что href и другие свойства, относящиеся к URL, доступны и для чтения и для записи. Следовательно, можно писать программы на JavaScript, которые динамически изменяют назначения гиперссылок! Вот легкомысленный фрагмент HTML, усиленный кодом JavaScript, который использует обработчик события объекта Link для записи в свойство href и создания ссылки, назначение которой случайно выбирается из набора других ссылок этого документа:

```
<a href="about:"
   onmouseover="status = 'Воспользуйся случаем... Нажми меня.'; return true;"
   onclick="this.href =
           document.links[Math.floor(Math.random()*document.links.length)];"
>
Случайная ссылка
</a>
```

Этот пример демонстрирует все рассмотренные нами возможности объекта Link: массив links[], применение обработчиков событий объекта Link и динамическую установку назначения ссылки. Обратите внимание, что пример устанавливает свойство href ссылки, но не использует свойство href случайно выбранной ссылки. Вместо этого для получения URL он полагается на метод toString() объекта Link.

14.8. Якорные элементы

Массив anchors[] объекта Document содержит объекты Anchor, представляющие именованные места в документе HTML, отмеченные тегом <a> и его атрибутом name. Массив anchors[] появился еще JavaScript 1.0, но объект Anchor – только в JavaScript 1.2. В предыдущих версиях элементы массива anchors[] неопределенны, и полезным является только свойство length.

Объект Anchor достаточно простой. Он определяет единственное стандартное свойство name, являющееся значением атрибута HTML name. Как и для объекта Link, текст, расположенный между тегами <a> и якорного элемента, содержится в свойстве text для Netscape 4 и в свойстве innerText в Internet Explorer 4. Ни одно из этих свойств не поддерживается стандартом W3C DOM, но в главе 17 мы увидим другие способы получения текстового содержимого элемента.

В примере 14.6 показана функция, создающая навигационное окно для указанного документа. Она показывает свойства text, innerText и name всех якорных элементов документа. Текст или имя якоря отображается в гиперссылке – щелчок по любому из якорных элементов приводит к прокрутке исходного окна для отображения этого якоря. Код в этом примере особенно полезен, если вы создаете ваши HTML-документы таким образом, что все заголовки секций заключены в якорные элементы. Например:

```
<a name="sect14.6"><h2>Якорный объект</h2></a>
```

Пример 14.6. Список всех якорных элементов

```
/*
 * ФАЙЛ: listanchors.js
 * Функция listanchors() получает документ в качестве аргумента и открывает
 * новое окно, которое служит для этого документа "окном навигации".
 * Новое окно отображает список всех якорных элементов документа. Выбор
 * любого якорного элемента в списке приводит к прокрутке документа до позиции
 * якоря. Документ не должен вызывать эту функцию для самого себя, пока он
 * или по крайней мере все его якорные элементы не будут полностью
 * проанализированы броузером.
 */
function listanchors(d) {
    // Открытие нового окна
    var newwin = window.open("", "navwin",
                             "menubar=yes,scrollbars=yes,resizable=yes," +
                             "width=600,height=300");
```

```
    // Задание заголовка окна
    newwin.document.writeln("<h1>Навигационное окно:<br>" +
                            document.title + "</h1>");
    // Список всех якорных элементов
    for(var i = 0; i < d.anchors.length; i++) {
        // Для каждого якорного элемента определяем отображаемый текст.
        // Сначала пытаемся получить текст между тегами <a> и </a>, используя
        // зависящее от броузера свойство. Если оно отсутствует, используем
        // name.
        var a = d.anchors[i];
        var text = null;
        if (a.text) text = a.text;                          // Netscape 4
        else if (a.innerText) text = a.innerText;           // IE 4+
        if ((text == null) || (text == '')) text = a.name;  // По умолчанию

        // Выводим текст в виде ссылки. Обратите внимание на использование
        // свойства location исходного окна.
        newwin.document.write('<a href="#' + a.name + '"' +
                              ' onclick="opener.location.hash="' + a.name +
                              '"; return false;">');
        newwin.document.write(text);
        newwin.document.writeln('</a><br>');
    }
    newwin.document.close();  // Никогда не забывайте закрывать документ!
}
```

14.9. Апплеты

Массив `applets[]` объекта Document содержит объекты, представляющие апплеты, встраиваемые в документ с помощью тега `<applet>` или `<object>`. Апплет – это переносимая, безопасная программа на Java, загружаемая через Интернет и исполняемая веб-броузером; Java поддерживается и в Netscape и в Internet Explorer (хотя IE 6 больше не включает поддержку Java по умолчанию).

И Netscape 3 и Internet Explorer 3 допускают вызов открытых методов и чтение/запись открытых свойств апплетов Java из JavaScript. (Как мы увидим в главе 22, Netscape также поддерживает намного более развитое двунаправленное взаимодействие между JavaScript и Java.) Все апплеты имеют несколько стандартных открытых методов, которые они наследуют от своих суперклассов, но наиболее интересные методы и свойства для каждого апплета различны. Если вы автор апплета, которым хотите управлять из JavaScript, то вы уже знаете, какие открытые методы и свойства он определяет. Если не вы написали апплет, то определить, что он позволяет сделать, можно, обратившись к его документации.

Ниже показано, как встроить Java-апплет в веб-страницу с помощью тега `<applet>` и затем вызвать методы `start()` и `stop()` этого апплета из обработчиков событий JavaScript:

```
<applet name="animation" code="Animation.class" width="500" height="200">
</applet>
<form>
<input type="button" value="Start" onclick="document.animation.start();">
<input type="button" value="Stop" onclick="document.animation.stop();">
</form>
```

Все апплеты определяют методы `start()` и `stop()`. В данном гипотетическом примере эти методы приводят к запуску и остановке анимации; определив HTML-форму, мы даем пользователю контроль над запуском и остановкой апплета. Обратите внимание, что мы указали атрибут `name` в теге `<applet>`, поэтому мы можем ссылаться на апплет по имени, а не как на нумерованный элемент массива `applets[]`.

Этот пример не является полноценной демонстрацией мощи JavaScript как языка сценариев для апплетов Java: методам Java, вызываемым из обработчиков событий JavaScript, не передаются аргументы, и методы не возвращают значений. На самом деле JavaScript может передавать числовые, строковые и логические значения в качестве аргументов методов Java и может принимать числовые, строковые и логические значения, возвращаемые из этих функций. (Как мы увидим в главе 22, Netscape также может передавать объекты JavaScript и Java в методы Java и получать объекты из них.) Автоматическое преобразование данных между JavaScript и Java позволяет организовать мощное взаимодействие между двумя программными средами. Например, апплет может реализовать метод, возвращающий строку кода JavaScript. JavaScript может затем воспользоваться методом `eval()` для исполнения этого кода.

Апплеты могут также реализовывать методы, работающие не с самим апплетом, а выступающие в качестве канала передачи данных между JavaScript и Java. Например, апплет может определить метод, вызывающий метод `System.getProperty()` для заданного строкового аргумента. Этот апплет позволяет JavaScript извлекать значения системных свойств Java и определять, например, версию Java, поддерживаемую броузером.

14.10. Вложенные данные

Массив `embeds[]` содержит объекты, представляющие данные (отличные от апплетов), вложенные в документ с помощью тега `<embed>` или `<object>`. Вложенные данные могут быть представлены в различной форме (аудио, видео, электронная таблица и т. д.). В броузере должна быть установлена (или ему должна быть доступна) соответствующая программа для просмотра, чтобы данные могли быть показаны пользователю. В Netscape специальные модули (plug-ins, плагины) отвечают за отображение вложенных данных. В Internet Explorer вложенные данные отображаются с помощью управляющих элементов ActiveX. И подключаемые модули и управляющие элементы ActiveX могут по необходимости автоматически загружаться по сети и устанавливаться.

В то время как все элементы массива `applets[]` представляют апплеты Java, элементы массива `embeds[]`, как правило, более разнообразны, и почти никаких общих сведений о них привести нельзя. Свойства и методы этих объектов зависят от конкретного подключаемого модуля или управляющего элемента ActiveX, используемого для отображения вложенных данных. Для получения информации о подключаемом модуле или управляющем элементе ActiveX необходимо обращаться к документации от производителя. Если подключаемый модуль или управляющий элемент поддерживает какие-либо возможности сценариев JavaScript, документация должна сообщать об этом и описывать свойства и методы, которые можно использовать из JavaScript. Например, документация к подключаемому модулю LiveVideo от Netscape говорит о том, что объект LiveVideo в массиве `embeds[]` поддерживает четыре метода: `play()`, `stop()`, `rewind()` и `seek()`. Имея эту информацию, можно писать простые сценарии, управляющие тем, как подключаемый модуль проигрывает видеоролик, который вы вставили на веб-страницу. Обратите внимание: несмотря на то что некоторые производители могут создавать подключаемые модули (для Netscape) и управляющие элементы ActiveX (для IE), определяющие одинаковые открытые API, это выполняется не всегда, и написание сценариев для работы с вложенными объектами обычно требует кода JavaScript, специфического для каждой платформы.

15

Формы и элементы форм

Как мы видели в примерах на протяжении этой книги, работа с формами HTML – это основная часть почти всех программ на JavaScript. В данной главе объясняются детали программирования на JavaScript с использованием форм. Предполагается, что вы уже в какой-то степени знакомы с созданием форм HTML и с содержащимися в них элементами ввода. Если нет, вы можете обратиться к хорошей книге по HTML Чака Муссиано (Chuck Musciano) и Билла Кеннеди (Bill Kennedy) «HTML: The Definitive Guide, 4th Edition», O'Reilly, 2000.[1] В части IV «Справочник по клиентскому JavaScript» наряду с синтаксисом JavaScript для форм и их элементов приведен и синтаксис HTML.

Те, кто уже знаком с программированием форм HTML на стороне сервера, заметят, что в случае форм с JavaScript все делается по-другому. В серверной модели форма с содержащимися в ней данными целиком отправляется на сервер. Акцент делается на обработке полного набора входных данных и динамическом генерировании веб-страницы в качестве отклика. В JavaScript используется совершенно иная модель программирования. В программах на JavaScript акцент делается не на передаче и обработке данных формы, а на обработке событий. Форма и все расположенные на ней элементы ввода имеют обработчики событий, позволяющие программировать реакцию на взаимодействие пользователя с формой. Если пользователь, например, щелкает по флажку, программа на JavaScript может получить уведомление через обработчик события и отреагировать на него, изменив значение, отображаемое в каком-либо другом элементе формы.

---

[1] Чак Муссиано, Билл Кеннеди «HTML и XHTML. Подробное руководство», 4-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2002.

В серверных программах форма HTML без кнопки Submit бесполезна (возможет вариант, когда форма содержит только одно текстовое поле ввода и позволяет нажать клавишу Return для передачи данных). С другой стороны, в JavaScript кнопка Submit не требуется (если, конечно, программа на JavaScript не работает совместно с программой на стороне сервера). В JavaScript форма может содержать произвольное количество кнопок с обработчиками событий, выполняющими при щелчке произвольное количество действий. В предыдущих главах мы видели некоторые возможные действия, вызываемые такими кнопками: замена одного изображения на другое, использование свойства `location` для загрузки и отображения новой веб-страницы, открытие нового окна броузера и динамическое создание нового HTML-документа в другом окне или фрейме. Как будет показано далее в этой главе, обработчик события в JavaScript может даже выполнить отправку данных формы.

Как мы видели, рассматривая примеры в этой книге, обработчики событий почти всегда представляют собой центральный элемент любой интересной программы на JavaScript. А чаще других обработчиков событий (за исключением обработчиков событий объекта Link) применяются те, которые связаны с формой и ее элементами. В этой главе вводится объект Form и различные объекты JavaScript, представляющие элементы формы. Завершается она примером, показывающим, как с помощью JavaScript проверить на стороне клиента введенные пользователем данные перед отправкой их программе, работающей на стороне веб-сервера.

15.1. Объект Form

Объект Form JavaScript представляет форму HTML. Формы всегда присутствуют в виде элементов массива `forms[]`, являющегося свойством объекта Document. Формы расположены в этом массиве в том же порядке, что и в документе. Следовательно, `document.forms[0]` ссылается на первую форму документа. На последнюю форму документа можно сослаться посредством следующего выражения:

```
document.forms[document.forms.length-1]
```

Наиболее интересное свойство объекта Form – это массив `elements[]`, содержащий объекты JavaScript (различных типов), представляющие различные элементы ввода в форме. Элементы этого массива также присутствуют в том же порядке, в каком они расположены в документе. На третий элемент второй формы документа в текущем окне можно сослаться так:

```
document.forms[1].elements[2]
```

Остальные свойства объекта Form менее важны. Свойства `action`, `encoding`, `method` и `target` соответствуют одноименным атрибутам тега `<form>`. Все эти свойства и атрибуты позволяют определять, как данные формы отправляются на веб-сервер и где будут отображаться результаты; следовательно, они

полезны только в том случае, когда форма действительно отсылается серверной программе. Описания этих свойств можно найти в части IV «Справочник по клиентскому JavaScript», а описания соответствующих атрибутов – в книге по HTML или по программированию CGI, например в книге Скотта Гулича (Scott Guelich), Шишира Гундаварама (Shishir Gundavaram) и Гюнтера Бирзнекса (Gunther Birznieks) «CGI Programming with Perl», 2nd Edition, O'Reilly, 2000.[1] Стоит отметить, что все свойства Form – это строки, доступные и для чтения и для записи, так что программа на JavaScript может динамически устанавливать их значения, чтобы изменить способ сохранения информации.

До JavaScript данные формы передавались при щелчке по специальной кнопке Submit, а для сброса значений элементов формы применялась специальная кнопка Reset. Объект Form JavaScript поддерживает два метода, `submit()` и (в JavaScript 1.1) `reset()`, служащие той же цели. Вызов метода `submit()` формы передает данные формы, а вызов `reset()` сбрасывает значения элементов формы.

В дополнение к методам `submit()` и `reset()` объект Form предоставляет обработчик события `onsubmit` для обнаружения отправки данных формы и (только в JavaScript 1.1) обработчик события `onreset` для обнаружения сброса значений полей формы. Обработчик `onsubmit` вызывается непосредственно перед передачей данных формы; он может отменить передачу, вернув `false`. Это дает возможность программе на JavaScript проверить введенные пользователем данные на наличие ошибок, чтобы предотвратить передачу серверу неполных или неверных данных. В конце этой главы мы увидим пример такой проверки. Обратите внимание, что обработчик `onsubmit` вызывается только при щелчке по кнопке Submit. Вызов метода `submit()` формы не приводит к вызову обработчика `onsubmit`.

Обработчик события `onreset` работает похожим образом. Он вызывается непосредственно перед очисткой формы и может предотвратить ее, вернув `false`. Это позволяет программе на JavaScript запросить подтверждение очистки данных, что может быть полезным в случае большой или подробной формы. Такое подтверждение можно запросить с помощью следующего обработчика события:

```
<form...
    onreset="return confirm('Действительно удалить ВСЕ данные и начать сначала?')"
>
```

Как и обработчик `onsubmit`, `onreset` вызывается только при щелчке по кнопке Reset. Вызов метода `reset()` формы не приводит к вызову `onreset`.

---

[1] Скотт Гулич, Шишир Гундаварам, Гюнтер Бирзнекс «CGI программирование на Perl», 2-е издание. – Пер. с англ. – СПб: Символ-Плюс», 2001.

15.2. Определение элементов формы

Элементы формы HTML – это элементарные объекты, из которых мы создаем графический интерфейс пользователя программы на JavaScript. На рис. 15.1 показана сложная форма, содержащая как минимум по одному элементу каждого из базовых типов. На тот случай, если вы еще не знакомы с элементами форм HTML, на рисунке приведены номера, идентифицирующие каждый тип элементов. Мы завершим этот раздел примером (пример 15.1), демонстрирующим код HTML и JavaScript, использованный для создания формы, изображенной на рис. 15.1, и подключения обработчиков событий ко всем элементам формы.

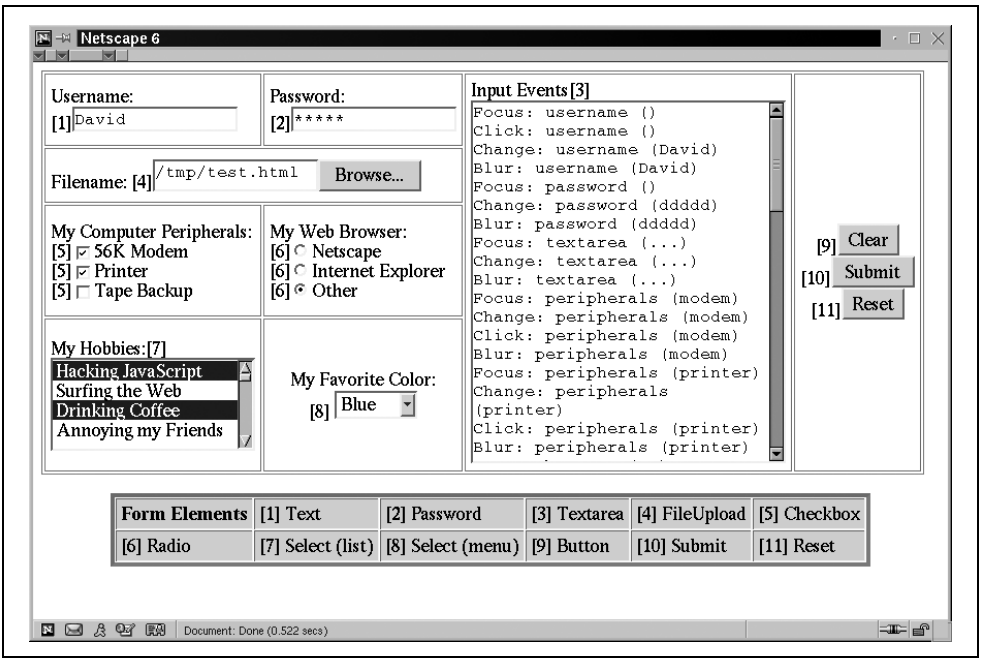

Рис. 15.1. *Элементы HTML-формы*

В табл. 15.1 перечислены типы элементов формы, доступных HTML-дизайнерам и программистам на JavaScript. В первом столбце таблицы расположены типы элементов формы, во втором – теги HTML, определяющие элементы этого типа, а в третьем столбце – значения свойства `type` для каждого типа элементов. Как мы видели, каждый объект Form содержит массив `elements[]`, в котором хранятся объекты, представляющие элементы формы. В каждом из этих элементов имеется свойство `type`, которое может применяться, чтобы отличить один тип элементов от другого. По свойству `type` неизвестного элемента формы код JavaScript может определить тип элемента и выяснить, что можно делать с этим элементом. (Мы увидим, как это делается, в примере 15.2 в конце этой главы.) И наконец, четвертый столбец таблицы предос-

тавляет краткое описание каждого элемента и наиболее важный или наиболее часто применяемый обработчик события для этого типа элемента.

Мы еще поговорим в этой главе об элементах формы. Подробно различные типы элементов описаны в части IV «Справочник по клиентскому JavaScript» (имена указаны в первом столбце табл. 15.1). Хотя для каждого типа элементов формы имеется отдельная справочная страница, обратите внимание, что большинство элементов создаются с помощью тега HTML `<input>` и на самом деле все являются объектами Input. На странице «Input» «Справочника по клиентскому JavaScript» перечислены возможности, общие для всех этих элементов, а страницы части IV, относящиеся к отдельным типам, содержат специфические сведения о работе с конкретным типом элементов формы. Обратите внимание, что имена «Button», «Checkbox» и т. д. из первого столбца таблицы могут не соответствовать реальным объектам в клиентской реализации JavaScript и что стандарт DOM не определяет какие-либо интерфейсы с этими именами. Тем не менее каждый из типов элементов формы имеет свой особенный внешний вид и поведение, и полезно, по крайней мере в «Справочнике по клиентскому JavaScript», рассматривать их как отдельные типы. В «Справочнике по W3C DOM» (часть V) вы найдете материал о формах и их элементах в разделах «HTMLFormElement», «HTMLInputElement», «HTMLTextAreaElement», «HTMLSelectElement» и «HTMLOptionElement».

Таблица 15.1. Элементы форм HTML

Объект	Тег HTML	Свойство type	Описание и события
Кнопка	`<input type="button">` или `<button type="button">`	«button»	Кнопка; `onclick`.
Флажок	`<input type="checkbox">`	«checkbox»	Флажок, не обладающий поведением переключателя; `onclick`.
Поле выбора файла	`<input type="file">`	«file»	Поле для ввода имени файла, загружаемого на веб-сервер; `onchange`.
Скрытое поле	`<input type="hidden">`	«hidden»	Данные, сохраняемые вместе с формой, но не видимые пользователю; обработчиков событий нет.
Вариант выбора	`<option>`	Нет	Один элемент объекта Select; обработчики событий подключаются к объекту Select, а не к отдельным объектам Option.
Поле ввода пароля	`<input type="password">`	«password»	Поле для ввода пароля (набранные символы невидимы); `onchange`.

Таблица 15.1. (продолжение)

Объект	Тег HTML	Свойство type	Описание и события
Переключатель	`<input type="radio">`	«radio»	Переключатель – одновременно может быть выбран только один; `onclick`.
Кнопка очистки формы	`<input type="reset">` или `<button type="reset">`	«reset»	Кнопка, очищающая значения формы; `onclick`.
Список с одиночным выбором	`<select>`	«select-one»	Список или выпадающее меню, из которого может быть выбран один элемент; `onchange` (см. также объект Option).
Список с множественным выбором	`<select multiple>`	«select-multiple»	Список, из которого может быть выбрано несколько элементов; `onchange` (см. также объект Option).
Кнопка передачи данных	`<input type="submit">` или `<button type="submit">`	«submit»	Кнопка, для передачи данных формы; `onclick`.
Текстовое поле ввода	`<input type="text">`	«text»	Однострочное поле ввода; `onchange`.
Текстовая область ввода	`<textarea>`	«textarea»	Многострочное поле ввода текста; `onchange`.

Теперь, взглянув на различные типы элементов формы и теги HTML, применяемые для их создания, посмотрим на код HTML из примера 15.1, использованный для создания формы, которая показана на рис. 15.1. Хотя большую часть примера занимает HTML, в нем также содержится код JavaScript, в котором определены обработчики событий каждого из элементов формы. Вы заметите, что обработчики событий не определяются как атрибуты HTML. Здесь ими являются функции JavaScript, присваиваемые свойствам объектов из массива `elements[]` формы. Все обработчики событий вызывают функцию `report()`, содержащую код, который работает с разными элементами формы. В следующем разделе этой главы описано все, что вам надо знать для понимания работы функции `report()`.

Пример 15.1. Форма HTML, содержащая все виды элементов

```
<form name="everything">                    <!-- HTML-форма все-в-одном... -->
 <table border="border" cellpadding="5">    <!-- в большой HTML-таблице -->
   <tr>
     <td>Username:<br>[1]<input type="text" name="username" size="15"></td>
     <td>Password:<br>[2]<input type="password" name="password" size="15"></td>
     <td rowspan="4">Input Events[3]<br>
       <textarea name="textarea" rows="20" cols="28"></textarea></td>
     <td rowspan="4" align="center" valign="center">
       [9]<input type="button" value="Clear" name="clearbutton"><br>
```

```
        [10]<input type="submit" name="submitbutton" value="Submit"><br>
        [11]<input type="reset" name="resetbutton" value="Reset"></td></tr>
    <tr>
      <td colspan="2">
        Filename: [4]<input type="file" name="file" size="15"></td></tr>
    <tr>
      <td>My Computer Peripherals:<br>
        [5]<input type="checkbox" name="peripherals" value="modem">56K Modem<br>
        [5]<input type="checkbox" name="peripherals" value="printer">Printer<br>
        [5]<input type="checkbox" name="peripherals" value="tape">Tape Backup</td>
      <td>My Web Browser:<br>
        [6]<input type="radio" name="browser" value="nn">Netscape<br>
        [6]<input type="radio" name="browser" value="ie">Internet Explorer<br>
        [6]<input type="radio" name="browser" value="other">Other</td></tr>
    <tr>
      <td>My Hobbies:[7]<br>
        <select multiple="multiple" name="hobbies" size="4">
          <option value="programming">Hacking JavaScript
          <option value="surfing">Surfing the Web
          <option value="caffeine">Drinking Coffee
          <option value="annoying">Annoying my Friends
        </select></td>
      <td align="center" valign="center">My Favorite Color:<br>[8]
        <select name="color">
          <option value="red">Red      <option value="green">Green
          <option value="blue">Blue       <option value="white">White
          <option value="violet">Violet<option value="peach">Peach
        </select></td></tr>
  </table>
</form>

<div align="center">        <!-- Еще одна таблица - ключ к приведенной выше -->
  <table border="4" bgcolor="pink" cellspacing="1" cellpadding="4">
    <tr>
      <td align="center"><b>Form Elements</b></td>
      <td>[1] Text</td>  <td>[2] Password</td>  <td>[3] Textarea</td>
      <td>[4] FileUpload</td> <td>[5] Checkbox</td></tr>
    <tr>
      <td>[6] Radio</td>  <td>[7] Select (list)</td>
      <td>[8] Select (menu)</td>  <td>[9] Button</td>
      <td>[10] Submit</td>  <td>[11] Reset</td></tr>
  </table>
</div>

<script>
// Эта обобщенная функция добавляет детали события к тексту в большом многострочном
// поле ввода вышеприведенной формы. Она вызывается из различных обработчиков.
function report(element, event) {
    var elmtname = element.name;
    if ((element.type == "select-one") || (element.type == "select-multiple")){
        value = " ";
        for(var i = 0; i < element.options.length; i++)
```

```
            if (element.options[i].selected)
                value += element.options[i].value + " ";
    }
    else if (element.type == "textarea") value = "...";
    else value = element.value;
    var msg = event + ": " + elmtname + ' (' + value + ')\n';
    var t = element.form.textarea;
    t.value = t.value + msg;
}

// Эта функция добавляет к каждому элементу в форме набор обработчиков событий.
// Она не проверяет, поддерживается ли в этом элементе данный обработчик,
// добавляются все обработчики событий. Обратите внимание, что обработчики событий
// вызывают приведенную выше функцию report(). Заметьте, что мы определяем
// обработчики событий, присваивая функцию свойствам объектов JavaScript, а не
// присваивая строки атрибутам HTML-элементов.
function addhandlers(f) {
    // Цикл по всем элементам формы
    for(var i = 0; i < f.elements.length; i++) {
        var e = f.elements[i];
        e.onclick = function() { report(this, 'Click'); }
        e.onchange = function() { report(this, 'Change'); }
        e.onfocus = function() { report(this, 'Focus'); }
        e.onblur = function() { report(this, 'Blur'); }
        e.onselect = function() { report(this, 'Select'); }
    }

    // Определяем несколько особых обработчиков событий для трех кнопок:
    f.clearbutton.onclick = function() {
        this.form.textarea.value=''; report(this,'Click');
    }
    f.submitbutton.onclick = function () {
        report(this, 'Click'); return false;
    }
    f.resetbutton.onclick = function() {
        this.form.reset(); report(this, 'Click'); return false;
    }
}
// И наконец, активизируем нашу форму, добавляя все возможные
// обработчики событий!
addhandlers(document.everything);
</script>
```

15.3. Сценарии и элементы формы

В предыдущем разделе перечислены элементы форм, предоставляемые языком HTML, и объясняется, как эти элементы встраиваются в HTML-документы. В этом разделе делается следующий шаг и показано, как можно работать с этими элементами из программы на JavaScript.

15.3.1. Назначение имен формам и элементам форм

У каждого элемента формы есть атрибут name, который должен быть установлен в теге HTML, если форма будет отправляться программе на стороне сервера. Хотя отправка данных формы, как правило, не представляет интереса для программ JavaScript, вы скоро увидите, что есть еще одна причина указывать значение этого атрибута.

У тега <form> также есть атрибут name, который можно установить. Этот атрибут не имеет никакого отношения к отправке форм. Он создан для удобства программистов JavaScript. Если в теге <form> определен атрибут name, то объект Form, создаваемый для данной формы, как обычно, сохраняется как элемент массива forms[] объекта Document, а также в собственном персональном свойстве объекта Document. Имя этого нового свойства представляет собой значение атрибута name. В частности, в примере 15.1 мы определили форму с помощью следующего тега:

```
<form name="everything">
```

Это позволило нам ссылаться на объект Form так:

```
document.everything
```

Часто это удобнее, чем нотация массива:

```
document.forms[0]
```

Кроме того, использование имени формы делает код позиционно независимым: он будет работать, даже если документ будет реорганизован таким образом, что формы будут расположены в нем в ином порядке.

У тегов HTML <img>, <applet> и других также есть атрибут name, действующий так же, как атрибут name тега <form>. Однако в формах назначение имен идет дальше, так как у всех элементов формы тоже есть собственный атрибут name. Когда вы даете имя элементу формы, вы создаете новое свойство объекта Form, ссылающееся на этот элемент. Именем этого свойства становится значение атрибута. Следовательно, вы можете ссылаться на элемент с именем zipcode, находящийся в форме с именем address, как на:

```
document.address.zipcode
```

Разумно выбрав имена, можно сделать синтаксис более элегантным, чем альтернативный способ, при котором индексы массивов жестко закодированы (и зависят от позиции):

```
document.forms[1].elements[4]
```

Чтобы в группе переключателей мог быть выбран только один элемент, всем входящим в группу элементам должны быть даны одинаковые имена. В примере 15.1 мы определили три переключателя, для которых значения атрибута name одинаковы: browser. Общепринятой, хотя и необязательной прак-

тикой также является назначение одинаковых атрибутов name группе взаимосвязанных флажков. Такое использование общих значений атрибута name работает естественным образом для программ на стороне сервера, но несколько неудобно для программ на стороне клиента. Однако эта проблема решена просто: когда несколько элементов формы имеют одинаковые атрибуты name, JavaScript просто помещает эти элементы в массив с указанным именем. Элементы массива располагаются в том порядке, в каком они присутствуют в документе. Поэтому на объекты Radio из примера 15.1 можно ссылаться так:

```
document.everything.browser[0]
document.everything.browser[1]
document.everything.browser[2]
```

15.3.2. Свойства элементов форм

У всех (или у большинства) элементов форм есть общие свойства, перечисленные ниже. Кроме того, у некоторых элементов есть специальные свойства, которые будут описаны позднее, при индивидуальном рассмотрении различных типов элементов форм.

type

Доступная только для чтения строка, идентифицирующая тип элемента формы. Значения этого свойства для каждого типа элементов форм перечислены в третьем столбце табл. 15.1.

form

Ссылка на объект Form, в котором содержится этот элемент.

name

Доступная только для чтения строка, указанная в HTML-атрибуте name.

value

Строка, доступная для чтения и для записи, задающая «значение», содержащееся в элементе формы или представляемое им. Эта строка отсылается на веб-сервер при передаче формы и только иногда представляет интерес для программ на JavaScript. Для элементов Text и Textarea это свойство содержит введенный пользователем текст. Для элементов Button это свойство задает текст, отображаемый внутри кнопки, который иногда требуется изменять из сценария. Свойство value для элементов Radio и Checkbox не редактируется и никак не отображается пользователю. Это просто строка, устанавливаемая HTML-атрибутом value, которая отсылается веб-серверу при передаче формы. Мы обсудим свойство value, когда позднее в этой главе будем рассматривать различные категории элементов формы.

15.3.3. Обработчики событий элементов форм

Большинство элементов форм поддерживают почти все перечисленные далее обработчики событий:

`onclick`

Вызывается при щелчке левой кнопкой мыши по данному элементу. Этот обработчик особенно полезен для кнопок и сходных с ними элементов формы.

`onchange`

Вызывается, когда пользователь изменяет значение, представляемое элементом, например вводит текст или выбирает элемент списка. Кнопки и сходные с ними элементы обычно не поддерживают этот обработчик события, так как у них нет значения, которое можно редактировать. Обратите внимание, что этот обработчик не вызывается, например, при каждом нажатии пользователем клавиши. Он вызывается, только когда пользователь изменил значение элемента и затем переместил фокус ввода к какому-либо другому элементу формы. То есть вызов этого обработчика события указывает на завершенное изменение.

`onfocus`

Вызывается, когда элемент формы получает фокус ввода.

`onblur`

Вызывается, когда элемент формы теряет фокус ввода.

В примере 15.1 показано, как определить обработчики событий для элементов формы. Пример разработан таким образом, что сообщает о событиях в момент их возникновения, перечисляя их в текстовой области ввода. Это дает возможность поэкспериментировать с элементами форм и вызываемыми ими обработчиками событий.

Важно знать, что в коде обработчика события ключевое слово `this` всегда ссылается на элемент документа, вызвавший данное событие. Во всех элементах формы имеется свойство `form`, ссылающееся на форму, в которой содержится элемент, поэтому обработчики событий элемента формы всегда могут обратиться к объекту Form как к `this.form`. Сделав еще один шаг, мы можем сказать, что обработчик события для одной формы может ссылаться на соседний элемент формы, имеющий имя `x`, как на `this.form.x`.

Обратите внимание, что в этом разделе перечислены только четыре обработчика событий, представляющие особую важность для элементов форм. Кроме них элементы форм также поддерживают различные обработчики (такие как `onmousedown`), которые поддерживаются (почти) всеми элементами HTML. Подробное обсуждение событий и обработчиков событий приведено в главе 19.

15.3.4. Кнопки

Элемент формы Button – один из наиболее часто используемых, так как он предоставляет понятный визуальный способ вызова пользователем какого-либо запрограммированного сценарием действия. Объект Button не имеет собственного поведения по умолчанию и не представляет никакой пользы без обработчика события `onclick` (или другого события). Свойство `value` эле-

мента Button управляет текстом, появляющимся на самой кнопке. В броузерах четвертого поколения это свойство можно установить, изменив текст (только чистый текст, а не HTML), присутствующий на кнопке, и это часто может быть полезно.

Обратите внимание, что гиперссылки предоставляют такой же обработчик события onclick, что и кнопки, и любой объект-кнопку можно заменить на ссылку, выполняющую при щелчке такое же действие. Когда вам требуется элемент, который выглядит как графическая кнопка, используйте кнопку. Когда действие, вызываемое обработчиком onclick, можно классифицировать как «переход по ссылке», используйте ссылку.

Элементы Submit и Reset очень похожи на элементы Button, но имеют связанные с ними действия по умолчанию (передача или очистка формы). Так как у этих элементов имеются действия по умолчанию, они могут быть полезны даже без обработчика событий onclick. С другой стороны, действия, выполняемые ими по умолчанию, определяют, что эти элементы полезнее для форм, отправляемых на веб-сервер, чем для чисто клиентских программ на JavaScript. Если обработчик события onclick возвращает false, стандартное действие этих кнопок не выполняется. Обработчик onclick элемента Submit позволяет проверить введенные в форме значения, но обычно это делается в обработчике onsubmit самой формы.

В HTML 4 можно создавать кнопки Button, Submit и Reset при помощи тега <button> вместо традиционного тега <input>. Тег <button> более гибок, так как отображает не просто текст, заданный атрибутом value, он отображает любое содержимое HTML (форматированный текст и/или изображения), присутствующее между <button> и </button>. Объекты Button, создаваемые тегом <button>, формально отличаются от созданных с помощью тега <input>, но имеют одинаковые значения поля type и в остальном их поведение довольно схоже. Основное отличие состоит в том, что поскольку тег <button> не использует значение свойства value для определения внешнего вида кнопки, внешний вид нельзя изменить путем установки свойства value. В этой книге мы употребляем термины «Button», «Submit» и «Reset» в отношении объектов, созданных как тегом <input>, так и тегом <button>.

15.3.5. Переключатели и флажки

Флажки и переключатели – это элементы, имеющие два визуально различных состояния: они могут быть либо отмечены, либо нет. Пользователь может изменить состояние такого элемента, щелкнув по нему. Переключатели объединяются в группы связанных элементов, которые имеют одинаковые значения HTML-атрибутов name. При выборе одного переключателя из группы выбор с других переключателей снимается. Флажки тоже часто составляют группы с одним значением атрибута name, и, ссылаясь на эти элементы по имени, необходимо помнить, что объект, на который вы ссылаетесь, представляет собой массив элементов с одинаковыми именами. В примере 15.1 имеется три объекта Checkbox с именами peripherals, и мы можем ссылаться на массив из трех этих элементов следующим образом:

```
document.everything.peripherals
```

Для ссылки на отдельный флажок мы должны указать его индекс в массиве:

```
document.everything.peripherals[0]  // Первый элемент формы
                                    // с именем "peripherals"
```

У флажков и переключателей есть свойство checked. Это доступное для чтения и записи логическое значение определяет, выбран ли в данный момент элемент. Свойство defaultChecked представляет собой доступное только для чтения логическое значение, содержащее значение HTML-атрибута checked; оно определяет, был ли элемент выбран при первой загрузке страницы.

Флажки и переключатели сами не отображают какой-либо текст и обычно выводятся вместе с прилегающим к ним текстом HTML (или, в HTML 4, со связанным тегом <label>). Это значит, что установка свойства value элемента Checkbox или Radio не изменяет внешнего вида элемента, как это происходит с элементами Button. Данное свойство можно установить, но это только изменит строку, отсылаемую на веб-сервер при передаче формы.

Когда пользователь щелкает по флажку или переключателю, элемент вызывает свой обработчик onclick для уведомления программы на JavaScript об изменении своего состояния. Более новые броузеры также вызывают для этих элементов обработчик onchange. Оба обработчика передают в основном одну и ту же информацию, но обработчик onclick характеризуется лучшей переносимостью.

15.3.6. Текстовые поля

Текстовое поле ввода представлено объектом Text и, пожалуй, чаще других элементов применяется в формах HTML и программах JavaScript. В это поле пользователь может ввести короткий, однострочный текст. Свойство value представляет текст, введенный пользователем. Установив это свойство, можно явно задать текст, который должен отображаться в поле. Обработчик события onchange вызывается, когда пользователь вводит новый текст или редактирует существующий, а затем указывает, что он завершил редактирование, переместив фокус ввода из текстового поля.

Текстовая область ввода (объект Textarea) очень похожа на текстовое поле ввода за исключением того, что разрешает пользователю ввести (и программе JavaScript вывести) многострочный текст. Текстовая область создается тегом <textarea>, при этом синтаксис существенно отличается от синтаксиса тега <input>, применяемого для создания текстового поля ввода. Тем не менее эти два типа элементов ведут себя очень похожим образом, и текстовая область ввода может считаться производной от HTMLInputElement, хотя формально это не так. Свойство value и обработчик события onchange текстовой области ввода можно применять точно так же, как это делается для текстового поля.

Элемент Password – это модификация текстового поля, отображающая символы звездочек вместо вводимого пользователем текста. Как следует из имени, эта особенность позволяет вводить пароли, не беспокоясь о том, что другие прочитают их через плечо. Элемент Password, как и текстовое поле ввода, вызывает свой обработчик события `onchange`, но на использование его свойства `value` накладываются некоторые ограничения (или ошибки). Некоторые старые броузеры (такие как Netscape 3) реализуют неэффективную меру безопасности, не дающую JavaScript читать значение, введенное пользователем в элемент Password. В других броузерах (таких как Netscape 4), значение `value` может быть установлено, но его установка не приводит ни к каким изменениям внешнего вида элемента формы. Обратите внимание, что элемент Password защищает введенные пользователем данные от любопытных глаз, но при отправке формы эти данные никак не шифруются (если только отправка не выполняется по безопасному HTTPS-соединению) и могут быть видимы при передаче по сети.

И наконец, объект FileUpload предназначен для ввода пользователем имени файла, который должен быть загружен на сервер. По существу, это текстовое поле ввода, совмещенное со встроенной кнопкой, выводящей диалоговое окно выбора файла. У элемента FileUpload, как и у текстового поля ввода, есть обработчик события `onchange`. Однако в отличие от текстового поля ввода, свойство `value` объекта FileUpload доступно только для чтения. Это не дает программам JavaScript, написанным со злым умыслом, обмануть пользователя, загрузив файл, который не предназначен для отправки на сервер.

Netscape 4, Internet Explorer 4 (и их более поздние версии) определяют обработчики событий `onkeypress`, `onkeydown` и `onkeyup` (заметьте, что они пока не являются частью стандарта DOM). Эти обработчики могут задаваться для любого объекта документа, но наиболее полезны (а в Netscape 4 единственно полезны) при назначении их текстовому полю ввода и подобным ему элементам, в которые пользователь вводит реальные данные. Можно вернуть `false` из обработчиков событий `onkeypress` или `onkeydown`, чтобы запретить обработку нажатой пользователем клавиши. Это может быть, например, полезно, когда требуется заставить пользователя вводить только цифры. Дополнительные подробности по этому и другим обработчикам событий, поддерживаемым всеми элементами HTML, можно найти в разделах «HTMLElement» частей IV и V («Справочник по клиентскому JavaScript» и «Справочник по W3C DOM» соответственно).

15.3.7. Списки

Элемент Select представляет собой набор вариантов (представленных элементами Option), которые могут быть выбраны пользователем. Броузеры обычно отображают элементы Select в виде выпадающих меню или списков. Элемент Select может работать двумя сильно отличающимися способами, и значение свойства `type` зависит от того, как он настроен. Если в теге `<select>` определен атрибут `multiple`, пользователь может выбрать несколько вариантов, и свойство `type` объекта Select равно «select-multiple». В противном слу-

чае, если атрибут multiple отсутствует, может быть выбран только один вариант, и свойство type равно «select-one».

В некотором отношении элемент «select-multiple» похож на набор флажков, а элемент «select-one» – на набор переключателей. Элемент Select отличается от них тем, что один элемент Select представляет полный набор вариантов. Эти варианты задаются в HTML с помощью тега <option> и представлены в JavaScript объектами Option, хранящимися в массиве options[] элемента Select. Элемент Select представляет набор вариантов, поэтому у него нет свойства value, как у других элементов формы. Вместо этого, как мы скоро обсудим, свойство value определяется в каждом объекте Option, содержащемся в элементе Select.

Когда пользователь выбирает вариант или отменяет выбор, элемент Select вызывает свой обработчик события onchange. Для элементов Select типа «select-one» значение доступного только для чтения свойства selectedIndex равно номеру выбранного в данный момент варианта. Для элементов «select-multiple» одного свойства selectedIndex недостаточно для представления полного набора выбранных вариантов. В этом случае для определения выбранных вариантов следует в цикле перебрать элементы массива options[] и проверить значения свойства selected каждого объекта Option.

Кроме свойства selected у элемента Option есть свойство text, задающее строку текста, которая отображается в элементе Select для данного варианта. Можно установить значение этого свойства, чтобы изменить отображаемый пользователю текст. Свойство value также представляет доступную для чтения и записи строку, задающую текст, который отсылается на веб-сервер при передаче данных формы. Даже если вы пишете чисто клиентскую программу и передача данных вашей формы никогда не происходит, свойство value (или соответствующий ей HTML-атрибут value) может быть полезным местом для хранения данных, которые вам потребуются при выборе пользователем определенного варианта. Обратите внимание, что элемент Option не определяет связанных с формой обработчиков событий; используйте вместо этого обработчик onchange соответствующего элемента Select.

Кроме установки свойства text объектов Option есть другие способы динамического изменения отображаемых в элементе Select вариантов. Можно обрезать массив элементов Option, установив options.length равным требуемому количеству вариантов, или удалить все объекты Option, установив options.length равным нулю. Предположим, у вас в форме с именем «address» есть объект Select с именем «country». Мы можем удалить все варианты из элемента следующим образом:

```
document.address.country.options.length = 0;  // Удаление всех опций
```

Мы также можем удалить отдельный объект Option элемента Select, записав в соответствующий элемент массива options[] значение null. Сделав это, вы удалите объект Option, и все элементы массива options[] с большими индексами автоматически сместятся для заполнения освободившейся ячейки:

```
// Удаление одного объекта Option из элемента Select
// Элемент Option, ранее расположенный в ячейке options[11],
// переместится в options[10] и т.п.
document.address.country.options[10] = null;
```

И наконец, элемент Option определяет конструктор `Option()`, который (в JavaScript 1.1 и более поздних версиях) можно использовать для динамического создания новых элементов Option. Таким образом, можно добавить новые варианты в элемент Select путем добавления их в конец массива `options[]`. Например:

```
// Создание нового объекта Option
var zaire = new Option("Zaire",  // Свойство text
                       "zaire",  // Свойство value
                       false,    // Свойство defaultSelected
                       false);   // Свойство selected

// Отображаем опцию в элементе Select, добавляя ее к массиву options:
var countries = document.address.country;  // Получаем объект Select
countries.options[countries.options.length] = zaire;
```

В HTML 4 для группировки связанных вариантов внутри элемента Select может применяться тег `<optgroup>`. В теге `<optgroup>` имеется атрибут `label`, задающий текст, который будет присутствовать в элементе Select. Однако, несмотря на свое видимое присутствие в списке, тег `<optgroup>` не может быть выбран пользователем, а объекты HTMLOptGroupElement никогда не присутствуют в массиве `options[]` элемента Select.

15.3.8. Скрытые элементы

Как следует из его имени, элемент Hidden не имеет визуального представления в форме. Он призван обеспечить передачу произвольного текста на сервер при передаче данных формы. Программы на стороне сервера используют элементы Hidden в качестве способа сохранения информации о состоянии, передаваемой обратно в серверную программу при передаче формы. Элементы Hidden не отображаются на странице, поэтому они не могут генерировать события и не имеют обработчиков событий. Свойство `value` позволяет читать и записывать текстовое значение, связанное с элементом Hidden, но, как правило, элементы Hidden практически не используются в программировании на клиентском JavaScript.

15.3.9. Элементы Fieldset

Стандарт HTML 4 добавляет к набору элементов, которые могут присутствовать в форме, новые теги `<fieldset>` и `<label>`. В IE 5 и более поздних версиях помещение тега `<fieldset>` на форму приводит к добавлению в массив `elements[]` соответствующего объекта. С точки зрения сценариев элементы Fieldset не столь интересны, как другие элементы форм, и в их объектах нет свойства `type`, как в других элементах форм. Поэтому наличие объектов

Fieldset в массиве elements[] кажется ошибочным решением разработчиков. Это подтверждается тем, что теги <label> не приводят к добавлению соответствующих объектов в массив elements[]. Разработчики броузеров Mozilla и Netscape 6 решили в этом отношении последовать примеру Microsoft для обеспечения совместимости с IE.

Это значит, что если вы определяете форму, содержащую элементы Fieldset, содержимое массива elements[] в новых, совместимых с HTML 4 броузерах, отличается от содержимого этого массива в броузерах, не поддерживающих HTML 4. В этой ситуации позиционные индексы в массиве elements[] не являются переносимыми, и необходимо определить атрибуты name для всех элементов формы и ссылаться на них по имени.

15.4. Пример верификации формы

Мы завершим обсуждение форм расширенным примером, демонстрирующим некоторые из рассмотренных нами концепций. В примере 15.2 показано как с помощью обработчика события onsubmit объекта Form выполнить проверку введенных данных, чтобы уведомить пользователя и отменить передачу данных формы, если они отсутствуют или введены неправильно. Изучив этот пример, можно вернуться к примеру 1.3, с которого мы начали эту книгу. Возможно, в коде этого примера теперь вы увидите больше смысла, так как стали экспертом в JavaScript!

В примере 15.2 определяется функция verify(), которая может выступать в качестве универсального средства проверки правильности заполнения форм. Она проверяет, не пусты ли обязательные поля. Кроме того, она может проверять, является ли числовое значение действительно числовым и попадает ли в указанный числовой диапазон. Эта функция verify()определяет тип элемента формы при помощи его свойства type. Также эта функция полагается на дополнительные пользовательские свойства для различия обязательных и необязательных полей и для задания допустимых диапазонов значений числовых полей. Обратите внимание, как функция читает свойство value поля ввода и при помощи свойства name выводит сообщения об ошибках.

На рис. 15.2 показан пример формы, в которой реализована эта схема проверки, и сообщение об ошибке, отображаемое, когда пользователь пытается сохранить некорректно заполненную форму.

Пример 15.2. Выполнение проверки формы

```
<script language="JavaScript1.1">
// Вспомогательная функция, возвращающая true, если строковая
// константа содержит только пробельные символы
function isblank(s) {
    for(var i = 0; i < s.length; i++) {
        var c = s.charAt(i);
        if ((c != ' ') && (c != '\n') && (c != '')) return false;
    }
    return true;
```

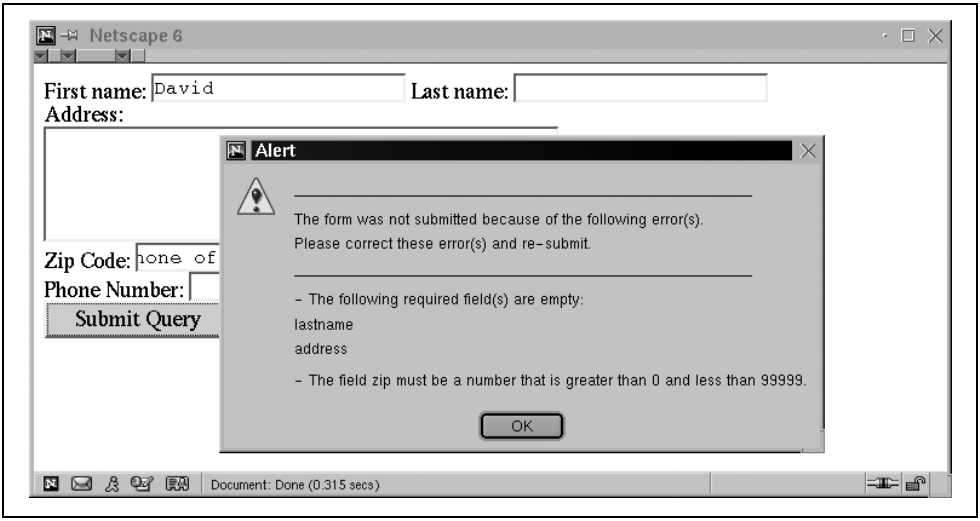

Рис. 15.2. *Форма, не прошедшая проверку*

```
}
// Это функция, выполняющая проверку формы. Она вызывается из обработчика
// событий onsubmit. Обработчик должен вернуть то значение, которое
// вернула эта функция.
function verify(f) {
    var msg;
    var empty_fields = "";
    var errors = "";

    // Выполняем цикл по элементам формы с поиском всех элементов Text и
    // Textarea, в которых не определено свойство "optional". Затем проверяем
    // на наличие пустых полей и создаем их список. Также, если для какого-либо
    // из этих элементов определены свойства "min" или "max", проверяем,
    // что введенные значения являются числами и попадают в нужный диапазон.
    // Если в элементе определено свойство "numeric", проверяем, что значение
    // является числом, но не проверяем его диапазон. Собираем вместе сообщения
    // об ошибках для всех полей, содержащих некорректные значения.
    for(var i = 0; i < f.length; i++) {
        var e = f.elements[i];
        if (((e.type == "text") || (e.type == "textarea")) && !e.optional) {
            // Сначала проверяем, не является ли поле пустым.
            if ((e.value == null) || (e.value == "") || isblank(e.value)) {
                empty_fields += "\n           " + e.name;
                continue;
            }

            // Теперь проверяем поля, которые предполагаются числовыми
            if (e.numeric || (e.min != null) || (e.max != null)) {
                var v = parseFloat(e.value);
                if (isNaN(v) ||
```

```
                    ((e.min != null) && (v < e.min)) ||
                    ((e.max != null) && (v > e.max))) {
                    errors += "- The field " + e.name + " must be a number";
                    if (e.min != null)
                        errors += " that is greater than " + e.min;
                    if (e.max != null && e.min != null)
                        errors += " and less than " + e.max;
                    else if (e.max != null)
                        errors += " that is less than " + e.max;
                    errors += ".\n";
                }
            }
        }
    }

    // Теперь, если есть ошибки, выводим сообщения и возвращаем
    // false для отмены подтверждения формы.
    // В противном случае возвращаем true.
    if (!empty_fields && !errors) return true;

    msg  = "_____________________________________________________\n\n"
    msg += "The form was not submitted because of the following error(s).\n";
    msg += "Please correct these error(s) and re-submit\n";
    msg += "_____________________________________________________\n\n"

    if (empty_fields) {
        msg += "- The following required field(s) are empty:"
                + empty_fields + "\n";
        if (errors) msg += "\n";
    }
    msg += errors;
    alert(msg);
    return false;
}
</script>

<!-----------------------------------------------------------------------
     Вот пример формы для тестирования нашей функции. Обратите внимание,
     что мы вызываем verify() из обработчика событий onsubmit и возвращаем
     то значение, которое было возвращено функцией. Также заметьте, что
     мы использовали обработчик onsubmit в качестве возможности установки
     свойств объектов формы, требуемых функцией verify() для выполнения
     проверки.
------------------------------------------------------------------------->
<form onsubmit="
    this.firstname.optional = true;
    this.phonenumber.optional = true;
    this.zip.min = 0;
    this.zip.max = 99999;
    return verify(this);
">
```

```
First name: <input type="text" name="firstname">
Last name: <input type="text" name="lastname"><br>
Address:<br><textarea name="address" rows="4" cols="40"></textarea><br>
Zip Code: <input type="text" name="zip"><br>
Phone Number: <input type="text" name="phonenumber"><br>
<input type="submit">
</form>
```

16

Сценарии и cookies

В объекте Document имеется не обсуждавшееся в главе 14 свойство под названием `cookie`. На первый взгляд кажется, что это свойство представляет собой простое строковое значение; однако свойство `cookie` управляет очень важным средством веб-броузера и имеет настолько большое значение, что ему посвящена отдельная глава.

16.1. Обзор cookies

Cookie – это небольшой объем именованных данных, сохраняемых веб-броузером и связанных с определенной веб-страницей или веб-сайтом.[1] Cookies играют роль памяти веб-броузера, чтобы сценарии и программы на стороне сервера могли на одной странице работать с данными, введенными на другой странице, или чтобы броузер мог вспомнить пользовательские параметры или другие переменные состояния, когда возвращается на страницу, посещенную им ранее. Cookies первоначально предназначались для CGI-программирования, и на низшем уровне реализованы как расширение протокола HTTP. Данные cookie автоматически передаются между веб-броузером и веб-сервером, так что CGI-сценарии на стороне сервера могут читать и записывать значения cookie, сохраняемые на стороне клиента. Как мы увидим,

---

[1] Термин «cookie» (булочка) не имеет особого значения, однако существовал прецедент, который ввел его в употребление. В туманных анналах истории компьютеров термин «cookie», или «magic cookie», использовался для обозначения небольшого объема данных, в особенности фрагмента привилегированных или секретных данных вроде пароля, подтверждающего подлинность или разрешающего доступ. В JavaScript cookies используются для сохранения состояния и могут служить веб-броузеру для установления подлинности. Однако в JavaScript cookies никак не шифруются и не обеспечивают безопасность.

JavaScript также может работать с cookie с помощью свойства `cookie` объекта Document.

Свойство `cookie` – это строковое свойство, позволяющее читать, создавать, изменять и удалять cookies, связанные с текущей веб-страницей. Хотя `cookie` с первого взгляда может показаться обычным доступным для чтения и записи строковым свойством, фактически его поведение более сложно. Читая значение `cookie`, мы получаем строку, содержащую имена и значения всех cookies, связанных с документом. Можно создавать, изменять и удалять cookies, устанавливая значение свойства `cookie`. В последующих разделах этой главы подробно объясняется, как это делается. Однако для того чтобы работа со свойством `cookie` была эффективной, надо побольше узнать о cookies и о том, как они работают.

Кроме имени и значения каждый cookie имеет четыре необязательных атрибута, управляющих временем его жизни, видимостью и безопасностью. Первый атрибут, `expires`, задает время жизни cookie. По умолчанию cookies являются временными – их значения сохраняются на период сеанса веб-броузера и теряются при закрытии его пользователем. Для того чтобы cookie сохранялся после окончания сеанса броузера, применяется атрибут `expires`, указывающий дату окончания действия cookie, – этот атрибут заставляет броузер сохранить cookie в локальном файле, чтобы он мог быть прочитан при следующем посещении пользователем веб-страницы. После того как дата окончания действия пройдет, файл cookie автоматически удаляется.

Второй атрибут cookie, `path`, задает веб-страницы, с которыми связан cookie. По умолчанию cookie связывается и доступен веб-странице, создавшей его, и любой другой веб-странице из того же каталога или любых его подкаталогов. Если, например, веб-страница *http://www.acme.com/catalog/index.html* создает cookie, то этот cookie также видим страницам *http://www.acme.com/catalog/order.html* и *http://www.acme.com/catalog/widgets/index.html*, но не видим для *http:// www.acme.com/about.html.*

Этого правила видимости, принятого по умолчанию, часто вполне достаточно. Тем не менее иногда значения cookie требуется использовать на всем многостраничном веб-сайте независимо от того, какая страница создала cookie. Например, если пользователь ввел свой адрес в форму на одной странице, этот адрес целесообразно сохранить как адрес по умолчанию при следующем возвращении пользователя на страницу и в совершенно другой форме на другой странице, где его просят ввести адрес для выставления счета. Для того чтобы это можно было сделать, для cookie задается значение `path`. Тогда любая страница с того же веб-сервера, содержащая указанное значение в своем URL, сможет совместно использовать cookie. Например, если для cookie, установленного страницей *http://www.acme.com/catalog/widgets/index.html*, для атрибута `path` задано значение «/catalog», этот cookie будет также виден для *http://www.acme.com/catalog/order.html*. А если путь установлен в «/», то cookie будет виден для любой страницы на веб-сервере *www.acme.com*.

По умолчанию cookies доступны только страницам, загружаемым с того веб-сервера, с которого были установлены cookies. Однако большим веб-сайтам может потребоваться возможность совместного использования cookies несколькими веб-серверами. Например, серверу *order.acme.com* может потребоваться прочитать значения cookie, установленных сервером *catalog.acme.com*. В этой ситуации поможет третий атрибут cookie – `domain`. Если cookie, созданный страницей с сервера *catalog.acme.com*, установил свой атрибут `path` равным «/», а атрибут `domain` – «.acme.com», этот cookie будет доступен всем веб-страницам серверов *catalog.acme.com*, *orders.acme.com* и любых других серверов в домене *acme.com*. Если атрибут `domain` для cookie не установлен, значением по умолчанию будет имя веб-сервера, на котором находится страница. Обратите внимание, что нельзя сделать домен cookie отличным от домена вашего сервера.

Четвертый и последний атрибут cookie – это логический атрибут с именем `secure`, определяющий, как значения cookie передаются по сети. По умолчанию cookie не являются безопасными, то есть передаются по обычному незащищенному HTTP-соединению. Однако если cookie помечен как безопасный, он передается, только когда броузер и сервер соединены с помощью протокола HTTPS или другого защищенного протокола.

Обратите внимание, что атрибуты `expires`, `path`, `domain` и `secure` не являются свойствами объекта JavaScript. Далее в этой главе мы увидим, как устанавливаются эти атрибуты cookie.

Те, кто интересуется техническими подробностями работы cookies, могут обратиться к спецификации на странице *http://www.netscape.com/newsref/std/cookie_spec.html*. Этот документ содержит исходную спецификацию HTTP cookies; в нем изложены низкоуровневые подробности, больше пригодные для CGI-программирования, чем для программирования на JavaScript. В следующих разделах рассматриваются установка и чтение значений cookie в JavaScript и установка атрибутов cookie `expires`, `path`, `domain` и `secure`.

16.2. Сохранение cookie

Для того чтобы связать временное значение cookie с текущим документом, достаточно установить свойство cookie равным строке следующего формата:

```
имя=значение
```

Например:

```
document.cookie = "version=" + escape(document.lastModified);
```

При следующем чтении свойства cookie сохраненная пара «имя/значение» будет включена в список cookie документа. Значения cookie не могут содержать точки с запятой, запятые или символы-разделители. По этой причине для кодирования значения перед сохранением его в cookie, возможно, потребуется использовать функцию JavaScript `escape()`. В этом случае при чтении значения cookie надо будет вызвать соответствующую функцию `unescape()`.

Cookie, записанный так, как описано выше, сохраняется для текущего сеанса веб-броузера, но теряется при выходе пользователя из броузера. Чтобы создать cookie, сохраняющийся между сеансами броузера, включите дату окончания действия, установив значение атрибута `expires`. Это можно сделать, установив значение свойства `cookie` равным строке следующего формата:

```
имя=значение; expires=дата
```

При установке даты подобным образом значение *дата* должно быть спецификацией даты в формате, создаваемом методом `Date.toGMTString()`. Например, создать cookie, сохраняющийся в течение года, можно посредством следующего кода:

```
var nextyear = new Date();
nextyear.setFullYear(nextyear.getFullYear() + 1);
document.cookie = "version=" + document.lastModified +
                  "; expires=" + nextyear.toGMTString();
```

Аналогичным образом можно установить атрибуты cookie `path`, `domain` и `secure`, дописав к значению cookie строки следующего формата перед его записью в свойство `cookie`:

```
; path=ïóòü
; domain=äîìåí
; secure
```

Для того чтобы изменить значение cookie, установите его значение снова, указав то же имя и новое значение. Задайте необходимые значения для `expires`, `path` и других атрибутов. Чтобы удалить cookie, установите его снова, указав то же имя, произвольное значение и дату окончания действия, которая уже прошла. Обратите внимание, что броузер не обязан удалять cookie немедленно, так что cookie может сохраниться в файле cookie броузера после даты окончания его действия.

16.2.1. Ограничения cookie

Cookie предназначены для редкого хранения небольших объемов данных. Они не являются универсальным средством коммуникации или механизмом передачи данных, и потому следует проявлять умеренность при их использовании. Веб-броузеры не обязаны сохранять более 300 cookies и 20 cookies на один веб-сервер (для всего веб-сервера, а не только для вашей страницы или сайта), или 4 Кбайт данных на один cookie (в этом ограничении учитываются и значение cookie и его имя). Наибольшее ограничение из перечисленных – лимит в 20 cookies на сервер. Для того чтобы предотвратить достижение этого ограничения, следует избегать использования отдельного cookie для каждой переменной состояния, которую вы хотите сохранить. Вместо этого можно записать несколько связанных между собой переменных состояния в один именованный cookie. В примере 16.1 далее в этой главе показано, как это сделать.

16.3. Чтение cookies

Когда свойство `cookie` используется в выражении JavaScript, возвращаемое им значение содержит все cookie, применимые к текущему документу.[1] Эта строка представляет собой список пар *имя=значение*, разделенных точками с запятой, где *имя* – это имя cookie, а *значение* – его строковое значение. Это значение не включает каких-либо атрибутов, которые могли быть установлены для cookie. Для получения значения cookie с определенным именем могут применяться методы `String.indexOf()` и `String.substring()` или (для разбиения строки на отдельные cookie) `String.split()`.

После извлечения значений cookie из свойства `cookie` их требуется интерпретировать, основываясь на том формате или кодировке, которые были указаны создателем cookie. Например, в одном cookie может храниться несколько единиц информации в полях, разделенных двоеточиями. В этом случае придется для извлечения различных фрагментов информации обратиться к соответствующим строковым методам. Не забудьте вызвать функцию `unescape()` для значения cookie, если оно было закодировано функцией `escape()`.

Следующий код показывает, как прочитать свойство cookie, извлечь из него отдельное значение cookie и потом использовать это значение:

```
// Чтение свойства cookie. Возвращает все cookie данного документа.
var allcookies = document.cookie;
// Поиск начала cookie с именем "version"
var pos = allcookies.indexOf("version=");

// Если мы нашли cookie с этим именем, извлекаем и используем его значение
if (pos != -1) {
    var start = pos + 8;                        // Начало значения cookie
    var end = allcookies.indexOf(";", start);  // Конец значения cookie
    if (end == -1) end = allcookies.length;
    var value = allcookies.substring(start, end);  // Извлекаем значение
    value = unescape(value);                       // Декодируем его

    // Теперь, получив значение cookie, мы можем его использовать.
    // В данном случае cookie было установлено равным дате изменения
    // документа, поэтому мы можем использовать это значение, чтобы узнать,
    // был ли документ изменен с момента последнего посещения пользователем.
    if (value != document.lastModified)
        alert("Этот документ был изменен с момента вашего последнего посещения");
}
```

[1] В Internet Explorer 3 свойство `cookie` работает только для объектов Document, извлеченных с помощью протокола HTTP. Документы, полученные из локальной файловой системы или с помощью других протоколов, например FTP, не могут иметь связанных с ними cookies.

Обратите внимание, что строка, полученная нами при чтении значения свойства cookie, не содержит какой-либо информации о различных атрибутах cookie. Свойство cookie позволяет установить эти атрибуты, но не позволяет прочитать их.

16.4. Пример работы с cookie

Пример 16.1 объединяет все аспекты cookie, которые мы до сих пор обсуждали. Сначала пример определяет класс Cookie. При создании объекта Cookie вы указываете объект Document, имя cookie и (необязательно) время окончания действия, путь, домен и логическое значение, указывающее, должен ли cookie быть защищен. После создания объекта Cookie можно установить в этом объекте произвольные строковые свойства. Значения этих свойств – это значения, которые будут храниться в cookie.

Класс Cookie определяет три метода. Метод store() обходит в цикле все определенные пользователем свойства объекта Cookie и конкатенирует их имена и значения в одну строку, выступает в качестве значения cookie. Метод load() объекта Cookie читает свойство cookie объекта Document для получения значений всех cookie данного документа. Он ищет в этой строке значение cookie с указанным именем и затем разбивает это значение на отдельные имена и значения, которые хранятся в качестве свойств объекта Cookie. И наконец, метод remove() объекта Cookie удаляет указанный cookie из документа.

За определением класса Cookie в примере 16.1 следует полезный и изящный способ использования cookie. Код этого примера несколько сложен, но стоит тщательного изучения. Можно начать с тестовой программы в конце примера, иллюстрирующей типичное применение класса Cookie.

Пример 16.1. Вспомогательный класс для работы с cookie

```
<script language="JavaScript1.1">
// Функция-конструктор: создает объект Cookie для указанного документа с
// указанным именем и необязательными атрибутами.
// Аргументы:
//   document: Объект Document, для которого сохраняется cookie. Обязательный.
//   name:     Строка, задающая имя cookie. Обязательный.
//   hours:    Необязательное число, задающее количество часов, по
//             прошествии которых cookie должен закончить свое действие.
//   path:     Необязательная строка, задающая атрибут path для cookie.
//   domain:   Необязательная строка, задающая атрибут domain для cookie.
//   secure:   Необязательное логическое значение, которое, будучи равно
//             true, требует защиты при передаче cookie.
//
function Cookie(document, name, hours, path, domain, secure)
{
    // Все предопределенные свойства этого объекта начинаются с символа '$', чтобы
    // отличить их от других свойств, являющихся значениями, предназначенными
    // для хранения в cookie.
    this.$document = document;
```

```
    this.$name = name;
    if (hours)
        this.$expiration = new Date((new Date()).getTime() + hours*3600000);
    else this.$expiration = null;
    if (path) this.$path = path; else this.$path = null;
    if (domain) this.$domain = domain; else this.$domain = null;
    if (secure) this.$secure = true; else this.$secure = false;
}

// Эта функция является методом store() объекта Cookie.
Cookie.prototype.store = function () {
    // Сначала выполняем цикл по свойствам объекта Cookie и собираем
    // значение cookie. Так как cookie используют знаки равенства и
    // точки с запятой в качестве разделителей, для индивидуальных
    // переменных состояния, хранящихся в одном значении cookie, мы
    // будем использовать двоеточния и амперсанды. Обратите внимание,
    // что мы кодируем каждую переменную состояния с помощью функции
    // escape() на тот случай, если она содержит знаки пунктуации или
    // другие недопустимые символы.
    var cookieval = "";
    for(var prop in this) {
        // Игнорирует свойства с именами, начинающимися с '$', а также методы
        if ((prop.charAt(0) == '$') || ((typeof this[prop]) == 'function'))
            continue;
        if (cookieval != "") cookieval += '&';
        cookieval += prop + ':' + escape(this[prop]);
    }

    // Теперь, когда у нас есть значение cookie, собираем вместе
    // полную строку cookie, включающую имя и различные атрибуты, указанные
    // при создании объекта Cookie.
    var cookie = this.$name + '=' + cookieval;
    if (this.$expiration)
        cookie += '; expires=' + this.$expiration.toGMTString();
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    if (this.$secure) cookie += '; secure';

    // Теперь сохраняем cookie, устанавливая «магическое» свойство
    // Document.cookie.
    this.$document.cookie = cookie;
}

// Эта функция является методом load() объекта Cookie
Cookie.prototype.load = function() {
    // Сначала получаем список всех cookie, относящихся к этому документу.
    // Мы делаем это путем чтения «магического» свойства Document.cookie.
    var allcookies = this.$document.cookie;
    if (allcookies == "") return false;

    // Теперь извлекаем из этого списка только cookie с указанным именем.
```

```
    var start = allcookies.indexOf(this.$name + '=');
    if (start == -1) return false;   // Для этой страницы cookie не определены
    start += this.$name.length + 1;  // Пропускаем имя и знак равенства
    var end = allcookies.indexOf(';', start);
    if (end == -1) end = allcookies.length;
    var cookieval = allcookies.substring(start, end);

    // После извлечения значения указанного cookie мы должны разбить
    // это значение на несколько отдельных переменных состояния.
    // Пары имя/значение отделяются друг от друга амперсандами, а
    // отдельные имена отделяются от значений двоеточиями. Для всех
    // этих операций мы используем метод split().
    var a = cookieval.split('&');  // Разбиваем на элементы массива пар "имя/значение"
    for(var i=0; i < a.length; i++) // Разбиваем каждую пару на два элемента массива
        a[i] = a[i].split(':');

    // После разбора значения cookie устанавливаем все имена и значения
    // переменных состояния в объекте в данном объекте Cookie. Обратите
    // внимание на использование функции unescape() для значения свойства;
    // она нужна потому, что при сохранении значений мы вызывали escape().
    for(var i = 0; i < a.length; i++) {
        this[a[i][0]] = unescape(a[i][1]);
    }

    // Все готово, поэтому возвращаем код успешного завершения
    return true;
}

// Эта функция является методом remove() объекта Cookie
Cookie.prototype.remove = function() {
    var cookie;
    cookie = this.$name + '=';
    if (this.$path) cookie += '; path=' + this.$path;
    if (this.$domain) cookie += '; domain=' + this.$domain;
    cookie += '; expires=Fri, 02-Jan-1970 00:00:00 GMT';

    this.$document.cookie = cookie;
}


//==================================================================
//  Приведенный выше код является определением класса Cookie.
//  Следующий код - это пример применения этого класса.
//==================================================================

// Создаем cookie, который мы будем использовать для сохранения
// состояния этой веб-страницы. Так как мы используем путь по умолчанию,
// этот cookie будет использоваться для всех веб-страниц, расположенных в
// том же каталоге или "ниже" его. Следовательно, cookie должен иметь
// имя, уникальное среди всех этих страниц. Обратите внимание, что мы
// устанавливаем дату окончания действия через десять дней после текущего
```

```
// момента.
var visitordata = new Cookie(document, "name_color_count_state", 240);

// Сначала пытаемся прочитать данные, хранящиеся в cookie. Если cookie
// не определен или не содержит нужных нам данных, запрашиваем данные
// у пользователя.
if (!visitordata.load() || !visitordata.name || !visitordata.color) {
    visitordata.name = prompt("Ваше имя:", "");
    visitordata.color = prompt("Ваш любимый цвет:", "");
}

// Отслеживаем, сколько раз пользователь посещал страницу:
if (visitordata.visits == null) visitordata.visits = 0;
visitordata.visits++;

// Сохраняем значения cookie, даже если они уже сохранялись, чтобы
// была установлена дата окончания действия, на 10 дней более поздняя, чем
// последний визит. Кроме того, снова сохраняем cookie, чтобы была
// сохранена обновленная переменная состояния visits.
visitordata.store();

// Теперь мы можем использовать прочитанные нами переменные состояния:
document.write('<font size="7" color="' + visitordata.color + '">' +
               Добро пожаловать, ' + visitordata.name + '!' +
               '</font>' +
               '<p>Вы посещали страницу ' + visitordata.visits + ' раз.');
</script>

<form>
<input type="button" value="Забудьте мое имя" onclick="visitordata.remove();">
</form>
```

17

Объектная модель документа

Объектная модель документа (Document Object Model, DOM) – это интерфейс прикладного программирования (API) для представления документа (такого как HTML-документ), а также обеспечения доступа к его элементам (таким как теги HTML и строки текста) и манипуляции ими. В броузерах, поддерживающих JavaScript, всегда определена объектная модель документа; DOM веб-броузера может, например, определять, что формы HTML-документа доступны через массив `forms[]` объекта Document.

В этой главе мы обсудим W3C DOM – стандартную объектную модель документа, определенную организацией World Wide Web Consortium и реализованную (по крайней мере, частично) броузерами Netscape 6, Internet Explorer 5 и Internet Explorer 6. Этот стандарт DOM[1] является полнофункциональным надмножеством объектных моделей традиционных броузеров. Он представляет документы HTML (и XML) в виде древовидной структуры и определяет свойства и методы для обхода, просмотра и изменения узлов дерева. Другие части стандарта определяют технологии для определения обработчиков событий для узлов документа, работы с таблицами стилей документа и работы с непрерывными областями документа.

Эта глава начинается с обзора стандарта DOM и затем описывает базовые части стандарта для работы с HTML-документами. После обсуждения основ стандарта следуют разделы, рассматривающие DOM-подобные средства Internet Explorer 4 и Netscape 4. Заканчивается глава обзором двух дополнительных частей стандарта DOM, тесно связанных с его ядром. В последую-

---

[1] Формально W3C выпускает «рекомендации». Однако эти рекомендации служат для тех же целей и имеют такой же вес, как и международные стандарты, и в этой книге называются «стандартами».

щих главах рассматриваются развитые средства DOM для работы с таблицами стилей и событиями.

17.1. Обзор DOM

DOM API не особенно сложен, но прежде чем переходить к обсуждению программирования с использованием DOM, следует разобраться с некоторыми вопросами архитектуры DOM.

17.1.1. Представление документов в виде деревьев

HTML-документы имеют иерархическую структуру, представленную в DOM в виде дерева. Узлы дерева представляют различные типы содержимого документа. В первую очередь, древовидное представление HTML-документа содержит узлы, представляющие элементы или теги, такие как `<body>` и `<p>`, и узлы, представляющие строки текста. HTML-документ также может содержать узлы, представляющие комментарии HTML.[1] Рассмотрим следующий простой HTML-документ.

```
<html>
  <head>
    <title>Sample Document</title>
  </head>
  <body>
    <h1>An HTML Document</h1>
    <p>This is a <i>simple</i> document.
  </body>
</html>
```

В DOM этот документ представлен деревом, изображенным на рис. 17.1.

Тем, кто еще не знаком с древовидными структурами в компьютерном программировании, полезно знать, что они заимствуют терминологию у генеалогических деревьев. Узел, расположенный непосредственно над данным узлом, называется *родительским* по отношению к данному узлу. Узлы, расположенные на один уровень ниже другого узла, являются дочерними по отношению к данному узлу. Узлы, находящиеся на том же уровне и имеющие того же родителя, называются *братьями*. Узлы, расположенные на любое число уровней ниже другого узла, являются его *потомками*. Родительские,

---

[1] DOM может также применяться для представления XML-документов, которые имеют значительно более сложный синтаксис, чем документы HTML. Древовидное представление таких документов может содержать узлы, представляющие ссылки на сущности XML, инструкции по обработке, разделы CDATA и т. д. Большинству программистов, имеющих дело с клиентским JavaScript, для работы с XML-документами DOM не потребуется, и хотя средства DOM, относящиеся к XML, рассматриваются в «Справочнике по W3C DOM», здесь о них не говорится.

прародительские и любые другие узлы, расположенные выше данного узла, являются его *предками*.

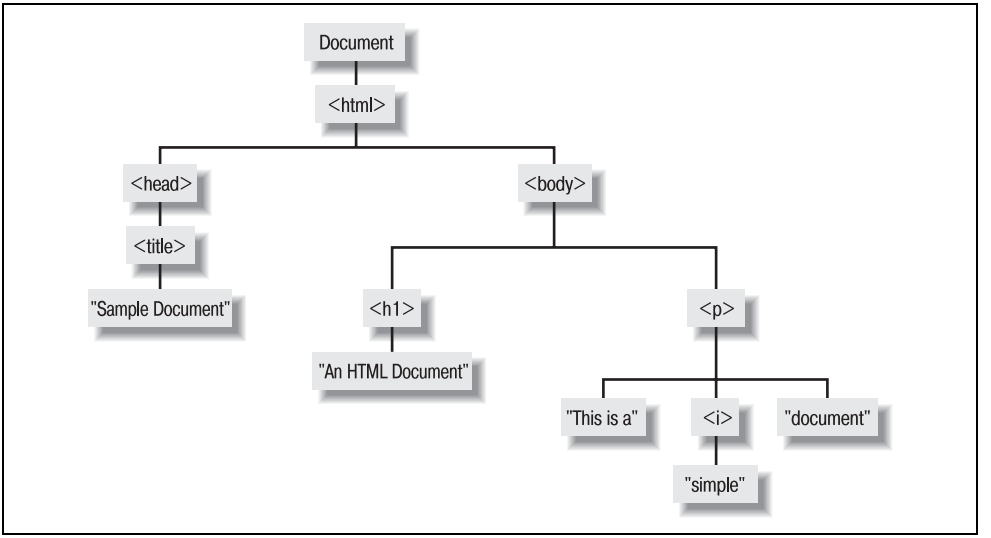

Рис. 17.1. *Древовидное представление HTML-документа*

17.1.2. Узлы

Древовидная структура DOM, изображенная на рис. 17.1, представляет собой дерево объектов Node различных типов. Интерфейс Node[1] определяет свойства и методы для перемещения по дереву и манипуляций с ним. Свойство `childNodes` объекта Node возвращает список дочерних узлов, свойства `firstChild`, `lastChild`, `nextSibling`, `previousSibling` и `parentNode` предоставляют способ обхода узлов дерева. Такие методы, как `appendChild()`, `removeChild()`, `replaceChild()` и `insertBefore()`, позволяют добавлять узлы в дерево документа и удалять их. Далее в этой главе мы увидим примеры применения этих свойств и методов.

17.1.2.1. Типы узлов

Типы узлов в дереве документа представлены специальными подынтерфейсами интерфейса Node. У любого объекта Node есть свойство `nodeType`, определяющее тип данного узла. Если свойство `nodeType` узла равно, например, константе `Node.ELEMENT_NODE`, значит, объект Node является также объектом Element, и можно использовать с ним все методы и свойства, определенные ин-

[1] Стандарт DOM определяет интерфейсы, а не классы. Те, кто не знаком с термином *интерфейс* в объектно-ориентированном программировании, могут рассматривать его как абстрактный вид класса. Позднее в этом обзоре DOM мы более подробно объясним различие между классом и интерфейсом.

терфейсом Element. В табл. 17.1 перечислены типы узлов, чаще всего встречающиеся в HTML-документах, и значения `nodeType` для каждого из них.

Таблица 17.1. Основные типы узлов

Интерфейс	Константа nodeType	Значение nodeType
Element	`Node.ELEMENT_NODE`	1
Text	`Node.TEXT_NODE`	3
Document	`Node.DOCUMENT_NODE`	9
Comment	`Node.COMMENT_NODE`	8
DocumentFragment	`Node.DOCUMENT_FRAGMENT_NODE`	11
Attr	`Node.ATTRIBUTE_NODE`	2

Корневым узлом дерева DOM является объект Document. Свойство `documentElement` этого объекта ссылается на объект Element, представляющий корневой элемент документа. Для HTML-документов это тег `<html>`, явно или неявно присутствующий в документе. (Помимо корневого элемента узел Document может иметь другие дочерние элементы, такие как объекты Comment.) Основная часть дерева DOM состоит из объектов Element, представляющих такие теги, как `<html>` и `<i>`, и объектов Text, представляющих строки текста. Если анализатор документа сохраняет комментарии, эти комментарии представляются в дереве как объекты DOM Comment. Неполная иерархия этих и других базовых интерфейсов DOM показана на рис. 17.2.

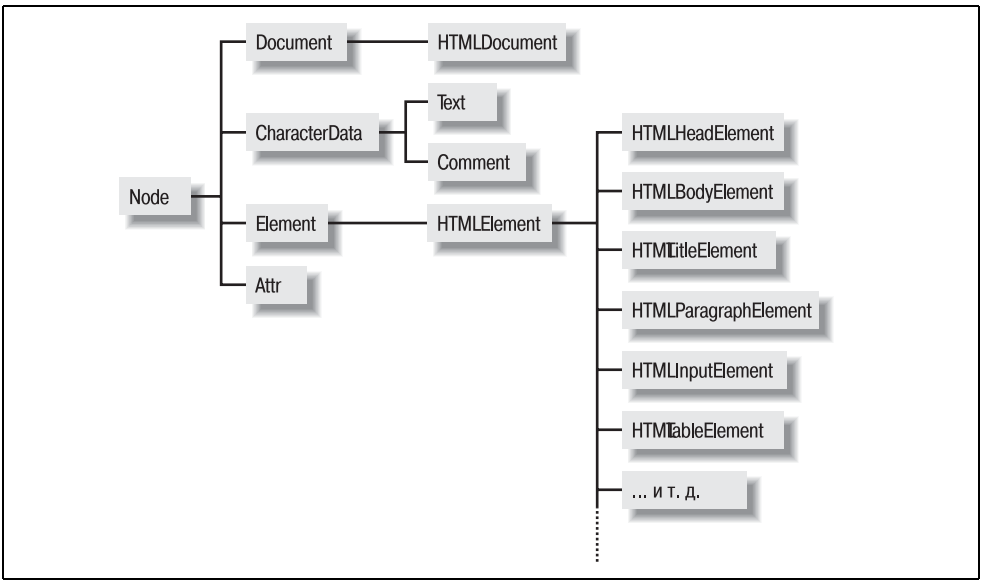

Рис. 17.2. *Неполная иерархия классов DOM API*

17.1.2.2. Атрибуты

Атрибуты элемента (например, атрибуты `src` и `width` тега `<img>`) могут быть прочитаны, установлены и удалены с помощью методов `getAttribute()`, `setAttribute()` и `removeAttribute()` интерфейса Element.

Другой, менее удобный способ работы с атрибутами – это метод `getAttributeNode()`, который возвращает объект Attr, представляющий атрибут и его значение. (Одной из причин выбора этой менее удобной технологии является наличие у интерфейса Attr свойства `specified`, позволяющего определять, указан ли данный атрибут в документе явно или для него принимается значение по умолчанию.) Интерфейс Attr присутствует на рис. 17.2 и представляет собою отдельный тип узла. Однако обратите внимание, что объекты Attr не присутствуют в массиве `childNodes[]` элемента и непосредственно не являются частью дерева документа, как узлы Element и Text. Спецификация DOM позволяет обращаться к узлам Attr через массив `attributes[]` интерфейса Node, но в Microsoft Internet Explorer определяется другой несовместимый массив `attributes[]`, что делает невозможным использование этого массива переносимым образом.

17.1.3. DOM HTML API

Стандарт DOM предназначен для работы как с XML, так и с HTML. Базовый DOM API – интерфейсы Node, Element, Document и другие – относительно универсален и применим к обоим типам документов. Стандарт DOM также включает интерфейсы, специфические для документов HTML. Как видно на рис. 17.2, HTMLDocument – это специфический для HTML подынтерфейс интерфейса Document, а HTMLElement – специфический для HTML подынтерфейс интерфейса Element. Кроме того, DOM определяет интерфейсы для многих элементов HTML, относящиеся к конкретным тегам. Эти интерфейсы, такие как HTMLBodyElement и HTMLTitleElement, обычно определяют набор свойств, отражающих атрибуты данного тега HTML.

Интерфейс HTMLDocument определяет различные свойства документа и методы, поддерживавшиеся броузерами до появления стандарта W3C. В их число входят свойство `location`, массив `forms[]` и метод `write()`, описанные в главах 13, 14 и 15.

Таблица 17.2. Простые теги HTML

`<abbr>`	`<acronym>`	`<address>`	`<b>`	`<bdo>`
`<big>`	`<center>`	`<cite>`	`<code>`	`<dd>`
`<dfn>`	`<dt>`	`<em>`	`<i>`	`<kbd>`
`<noframes>`	`<noscript>`	`<s>`	`<samp>`	`<small>`
`<span>`	`<strike>`	`<strong>`	`<sub>`	`<sup>`
`<tt>`	`<u>`	`<var>`		

Интерфейс HTMLElement определяет свойства `id`, `style`, `title`, `lang`, `dir` и `className`, обеспечивающие удобный доступ к значениям атрибутов `id`, `style`,

title, lang, dir и class, которые могут применяться со всеми тегами HTML. Теги HTML из табл. 17.2 не принимают никаких атрибутов, кроме шести только что перечисленных, и потому полностью представимы интерфейсом HTMLElement.

Для всех остальных тегов HTML в части спецификации DOM, относящейся к HTML, определяются специальные интерфейсы. Для многих тегов HTML эти интерфейсы не делают ничего, кроме предоставления набора свойств, соответствующих атрибутам HTML. Например, тегу <ul> соответствует интерфейс HTMLU ListElement, а для тега <body> есть соответствующий интерфейс HTMLBodyElement. Поскольку эти интерфейсы просто определяют свойства, стандартизованные в HTML, они не документируются в этой книге подробно. Можно спокойно предположить, что объект HTMLElement, представляющий определенный тег HTML, имеет свойства для каждого из стандартных атрибутов этого тега (см. соглашения о назначении имен в следующем разделе). Обратите внимание, что стандарт DOM определяет свойства для атрибутов HTML для удобства создателей сценариев. Общий (и, пожалуй, предпочтительный) способ чтения и установки значений атрибутов предоставляют методы getAttribute() и setAttribute() объекта Element.

Некоторые из интерфейсов, описанных в HTML DOM, определяют дополнительные свойства или методы, отличные от тех, что соответствуют значениям атрибутов HTML. Например, интерфейс HTMLInputElement определяет методы focus() и blur(), а интерфейс HTMLFormElement – методы submit() и reset() и свойство length. Такие методы и свойства обычно присутствовали до стандартизации DOM и были сделаны частью стандарта для обратной совместимости с принятой практикой программирования. Подобные интерфейсы документированы в «Справочнике по W3C DOM» (часть V). Кроме того, информацию о частях этих интерфейсов, относящихся к «принятой практике», можно найти в части IV «Справочник по клиентскому JavaScript», хотя зачастую эта информация приведена под именем, использовавшимся до стандартизации DOM, – так, HTMLFormElement и HTMLInputElement в «Справочнике по клиентскому JavaScript» описаны в разделах «Form» и «Input».

17.1.3.1. Соглашения о назначении имен для HTML

При работе со специфическими для HTML частями стандарта DOM необходимо иметь в виду некоторые простые соглашения о назначении имен. Имена свойств, специфических для интерфейсов HTML, начинаются со строчных букв. Если имя свойства состоит из нескольких слов, первые буквы второго и последующих слов являются прописными. Таким образом, атрибут maxlength тега <input> транслируется в свойство maxLength интерфейса HTMLInputElement.

Когда имя атрибута HTML конфликтует с ключевым словом JavaScript, для разрешения конфликта к имени добавляется префикс «html». Например, атрибут for тега <label> транслируется в свойство htmlFor интерфейса HTMLLabelElement. Исключение из этого правила составляет атрибут class (кото-

рый может быть указан для любого элемента HTML) – он транслируется в свойство `className`[1] интерфейса HTMLElement.

17.1.4. Уровни и возможности DOM

Имеются две версии, или два «уровня», стандарта DOM. DOM Level 1 был стандартизован в октябре 1998 года. Он определяет базовые интерфейсы DOM, такие как Node, Element, Attr и Document, а также различные интерфейсы, специфические для HTML. DOM Level 2 был стандартизован в ноябре 2000 года.[2] Кроме некоторых изменений в базовых интерфейсах, эта версия DOM была сильно расширена за счет определения стандартных API для работы с событиями документа и каскадными таблицами стилей (CSS), а также с целью предоставления дополнительных инструментальных средств для работы с непрерывными областями документов. На момент написания этой книги рабочая группа DOM в W3C занимается стандартизацией DOM Level 3. Кроме того, иногда вы можете встретить упоминание о DOM Level 0. Этот термин не относится к какому-либо формальному стандарту, а служит для неформальной ссылки на общие средства объектных моделей документа, реализованных в Netscape и Internet Explorer до стандартизации консорциумом W3C.

В DOM Level 2 стандарт стал модульным. Базовый модуль, определяющий основную древовидную структуру документа с помощью (среди прочих) интерфейсов Document, Node, Element и Next, – это единственный модуль, являющийся обязательным. Все остальные модули не обязательны и могут либо поддерживаться, либо нет, в зависимости от реализации. Реализация DOM в веб-броузере, очевидно, должна поддерживать модуль HTML, так как веб-документы пишутся на HTML. Броузеры, поддерживающие таблицы стилей CSS, обычно поддерживают модули StyleSheets и CSS, поскольку (как мы увидим в главе 18) стили CSS играют ключевую роль в программировании Dynamic HTML. Аналогично, так как большинство интересных программ JavaScript требует наличия средств обработки событий, можно предполагать поддержку веб-броузерами модуля Events спецификации DOM. К сожалению, модуль Events был лишь недавно определен спецификацией DOM Level 2 и не имел широкой поддержки на момент написания этой книги. Полный список модулей DOM Level 2 мы увидим в следующем разделе.

17.1.5. Соответствие DOM

На момент написания этой книги не существовало броузера, полностью соответствующего стандарту DOM. Последние релизы Mozilla ближе всего по-

---

[1] Имя `className` обманчиво, так как кроме указания имени одного класса это свойство (и представляемый им атрибут HTML) может содержать список имен классов, разделенных пробелами.

[2] За исключением частей стандарта, относящихся к HTML и к ноябрю 2001 года все еще находившихся в стадии «рабочего проекта». К счастью, текущий рабочий проект считается стабильным и включает лишь небольшие изменения (описанные в этой книге) по сравнению с относящимися к HTML частями стандарта Level 1.

дошли к этому, и полная совместимость с DOM Level 2 является целью проекта Mozilla. Броузер Netscape 6.1 соответствует большинству важных модулей Level 2, а Netscape 6.0 отличает достаточно хорошая совместимость, но с некоторым пробелами. Internrt Explorer 6 в основном совместим (по крайней мере, за одним неприятным исключением) с DOM Level 1, но не поддерживает многие модули Level 2, в частности, модуль Events, который обсуждается в главе 19. Internet Explorer 5 и 5.5 имеют значительные пробелы в совместимости, но достаточно хорошо поддерживают ключевые методы DOM Level 1, чтобы запускать большинство примеров из этой главы. В версии IE для Macintosh реализована значительно более полная поддержка DOM, чем в IE 5 для Windows.

Кроме Mozilla, Netscape, Internet Explorer и несколько других броузеров предлагают как минимум частичную поддержку DOM. Количество доступных броузеров стало слишком велико, а изменения в сфере поддержки стандартов происходят слишком быстро, чтобы даже пытаться в этой книге определенно утверждать, какие средства DOM поддерживает конкретный броузер. Следовательно, при определении совместимости реализации DOM в любом конкретном броузере вам придется полагаться на другие источники информации.

Одним из источников информации о совместимости является сама реализация. В совместимой реализации свойство `implementation` объекта Document ссылается на объект DOMImplementation, определяющий метод под названием `hasFeature()`. Посредством этого метода (если он существует) можно получить сведения о наличии поддержки определенного модуля (или возможностей) стандарта DOM. Например, определить, поддерживает ли реализация DOM в веб-броузере базовые интерфейсы DOM Level 1 для работы с документами HTML, можно с помощью следующего кода:

```
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("html", "1.0")) {
    // Броузер заявляет о поддержке базовых интерфейсов Level 1
    // и интерфейсов HTML
}
```

Метод `hasFeature()` принимает два аргумента: первый – это имя проверяемого модуля, а второй – номер версии в виде строки. Он возвращает `true`, если указанная версия указанного модуля поддерживается. В табл. 17.3 перечислены пары «название/номер версии», определенные в стандартах DOM Level 1 и Level 2. Обратите внимание, что названия модулей нечувствительны к регистру. В четвертом столбце таблицы указано, какие другие модули требуются для поддержки данного модуля, и, следовательно, их наличие подразумевается в случае возвращения методом значения `true`. Например, если `hasFeature()` указывает, что поддерживается модуль MouseEvents, это подразумевает, что также поддерживается модуль UIEvents, который, в свою очередь, подразумевает поддержку модулей Events, Views и Core.

Таблица 17.3. Модули, совместимость с которыми можно проверить методом hasFeature( )

Название модуля	Версия	Описание	Подразумевает
HTML	1.0	Базовые и HTML-интерфейсы Level 1	
XML	1.0	Базовые и XML-интерфейсы Level 1	
Core	2.0	Базовые интерфейсы Level 2	
HTML	2.0	HTML-интерфейсы Level 2	Core
XML	2.0	XML-интерфейсы Level 2	Core
Views	2.0	Интерфейс AbstractView	Core
StyleSheets	2.0	Универсальный обход таблицы стилей	Core
CSS	2.0	Стили CSS	Core, Views
CSS2	2.0	Интерфейс CSS2Properties	CSS
Events	2.0	Инфраструктура для обработки событий	Core
UIEvents	2.0	События пользовательского интерфейса (плюс модули Events и Views)	Events, Views
MouseEvents	2.0	События мыши	UIEvents
HTMLEvents	2.0	События HTML	Events
MutationEvents	2.0	События изменения документа	Events
Range	2.0	Интерфейсы областей документа	Core
Traversal	2.0	Интерфейсы для обхода документа	Core

В Internet Explorer 6 (под Windows) `hasFeature()` возвращает `true` только для модуля «HTML» и версии «1.0». Он не сообщает о совместимости с любыми другими модулями, перечисленными в табл. 17.2. (Хотя, как мы увидим в главе 18, он поддерживает большинство основных применений модуля CSS2.) В Netscape 6.1 `hasFeature()` возвращает `true` для большинства имен модулей и номеров версий, за исключением модулей Traversal и MutationEvents. Метод возвращает `false` для модулей Core и CSS2 версии 2.0, указывая на неполную совместимость (даже несмотря на то, что поддержка этих модулей очень хороша).

В этой книге документируются интерфейсы, составляющие все модули DOM, перечисленные в табл. 17.2. Модули Core, HTML, Traversal и Range рассматриваются в этой главе. Модули StyleSheets, CSS и CSS2 рассматриваются в главе 18, а различные модули, относящиеся к событиям (за исключением MutationEvents) – в главе 19. Часть V «Справочник по W3C DOM» содержит полное описание всех модулей.

Метод `hasFeature()` не абсолютно надежен. Как отмечалось выше, IE 6 сообщает о совместимости Level 1 со средствами HTML, даже несмотря на то, что в этой совместимости есть некоторые проблемы. С другой стороны, Netscape 6.1 сообщает о несовместимости с Level 2 Core, даже несмотря на то, что этот броузер почти совместим с этим модулем. В обоих случаях нужны более подробные сведения о том, что именно совместимо, а что нет. Но объем этой ин-

формации слишком велик, и она слишком изменчива, чтобы включать ее в печатное издание.

Те, кто активно занимается веб-разработкой, несомненно, уже знают или скоро узнают о многих специфических для броузеров деталях совместимости. Кроме того, в Интернете есть ресурсы, которые могут оказаться полезными. Самое важное, что организация W3C (в сотрудничестве с Национальным институтом стандартов и технологии США) работает над созданием набора с открытыми исходными текстами для тестирования реализаций DOM. На момент написания этой книги разработка тестового набора только начиналась, но он должен стать неоценимым средством тонкой проверки на совместимость реализации DOM. Подробности можно найти на сайте *http://www.w3c.org/DOM/Test/*.

Организация Mozilla имеет несколько тестовых наборов для различных стандартов, в том числе для DOM Level 1 (доступных на странице *http://www.mozilla.org/quality/browser_sc.html*). Компания Netscape опубликовала тестовый набор, включающий некоторые тесты для DOM Level 2 (доступный на странице *http://developer.netscape.com/evangelism/tools/testsuites/*). Netscape также опубликовала предвзятое (и устаревшее) сравнение совместимости с DOM ранних релизов Mozilla и IE 5.5 (доступное на странице *http://home.netscape.com/browsers/future/standards.html*). И наконец, вы также можете найти информацию о совместимости и соответствии стандартам на независимых сайтах в Интернете. Один из достойных упоминания сайтов публикует Петер-Пауль Кох (Peter-Paul Koch). Ссылку на таблицу совместимости DOM можно найти на его главной странице, посвященной JavaScript (*http://www.xs4all.nl/~ppk/js/*).

17.1.5.1. Совместимость с DOM в Internet Explorer

Поскольку IE – наиболее широко используемый веб-броузер, несколько особых замечаний о его совместимости со спецификациями DOM будут здесь уместны. IE 5 и более поздние версии достаточно хорошо поддерживают базовые и HTML-средства Level 1, чтобы запускать примеры из этой главы, и достаточно хорошо поддерживают ключевые возможности CSS Level 2 для того, чтобы запускать большинство примеров из главы 18. К сожалению, IE 5, 5.5 и 6 не поддерживают модуль Events из DOM Level 2, даже несмотря на то, что корпорация Microsoft участвовала в определении этого модуля и имела достаточно времени для его реализации в IE 6. Как мы увидим в главе 19, обработка событий играет ключевую роль в клиентском JavaScript, и отсутствие в IE поддержки стандартной модели обработки событий затрудняет разработку развитых клиентских веб-приложений.

Хотя IE 6 заявляет (через свой метод `hasFeature()`) о поддержке базовых и HTML-интерфейсов стандарта DOM Level 1, фактически эта поддержка не полна. Наиболее вопиющая проблема, с которой вы, скорее всего, столкнетесь, – небольшая, но неприятная: IE не поддерживает константы типов узлов, определяемых в интерфейсе Node. Вспомните, что каждый узел в документе имеет свойство `nodeType`, задающее тип данного узла. Спецификация

DOM также утверждает, что интерфейс Node определяет константы, представляющие каждый из определяемых им типов узлов. Например, константа `Node.ELEMENT_NODE` представляет узел Element. В IE (по крайней мере, до версии 6 включительно) эти константы просто не существуют.

Примеры в этой главе были изменены, чтобы обойти это препятствие, и содержат целые литералы вместо соответствующих символических констант. Например:

```
if (n.nodeType == 1 /*Node.ELEMENT_NODE*/)
                        // Проверяем, что n является объектом Element
```

Хороший стиль программирования требует, чтобы в код помещались константы, а не жестко кодируемые целые литералы, и те, кто захочет сделать код переносимым, могут включить в программу следующий код для определения констант, если они отсутствуют:

```
if (!window.Node) {
    var Node = {            // Если объект Node отсутствует, определяем
        ELEMENT_NODE: 1,    // его со следующими свойствами и значениями.
        ATTRIBUTE_NODE: 2,  // Обратите внимание, что это только типы узлов
        TEXT_NODE: 3,       // HTML. Для XML-узлов вам нужно определить
        COMMENT_NODE: 8,    // здесь другие константы.
        DOCUMENT_NODE: 9,
        DOCUMENT_FRAGMENT_NODE: 11
    }
}
```

17.1.6. Независимые от языка интерфейсы DOM

Хотя стандарт DOM появился благодаря желанию иметь общий API для программирования динамического HTML, модель DOM интересна не только веб-программистам. На самом деле сейчас этот стандарт наиболее интенсивно используется серверными программами на Java и C++ для анализа документов XML и манипуляции ими. Из-за наличия многих вариантов использования стандарт DOM был определен как независимый от языка. В данной книге описывается только привязка DOM API к JavaScript, но необходимо иметь в виду и некоторые другие моменты. Во-первых, заметьте, что свойства объекта в привязке к JavaScript обычно соответствуют паре методов get/set в привязке к другим языкам. Следовательно, когда программист, пишущий на Java, спрашивает вас о методе `getFirstChild()` интерфейса Node, надо понимать, что в JavaScript привязка Node API не определяет метода `getFirstChild()`. Вместо этого она просто определяет свойство `firstChild`, и чтение этого свойства в JavaScript эквивалентно вызову метода `getFirstChild()` в Java.

Другая важная особенность привязки DOM API к JavaScript в том, что некоторые объекты DOM ведут себя как массивы JavaScript. Если интерфейс определяет метод с именем `item()`, объекты, реализующие этот интерфейс, ведут себя так же, как доступные только для чтения массивы с числовым индексом. Предположим, что в результате чтения свойства `childNodes` узла по-

лучен объект NodeList. Отдельные объекты Node из списка можно получить, во-первых, передав номер нужного узла методу `item()`, а во-вторых, рассматривая объект NodeList как массив и обращаясь к нему по индексу. Следующий код иллюстрирует эти две возможности:

```
var n = document.documentElement;  // Это объект Node.
var children = n.childNodes;       // Это объект NodeList.
var head = children.item(0);       // Это один из способов
                                   // использования NodeList.
var body = children[1];            // Но есть более простой способ!
```

Аналогично, если у объекта DOM есть метод `namedItem()`, передача строки этому методу – то же самое, что использование строки как индекса массива. Например, следующие строки кода представляют собой эквивалентные способы доступа к элементу формы:

```
var f = document.forms.namedItem("myform");
var g = document.forms["myform"];
var h = document.forms.myform;
```

Стандарт DOM может использоваться различными способами, поэтому разработчики стандарта аккуратно определили DOM API таким образом, чтобы это не ограничивало возможность реализации API другими разработчиками. В частности, стандарт DOM определяет интерфейсы вместо классов. В объектно-ориентированном программировании класс – это фиксированный тип данных, который должен быть реализован в точном соответствии со своим определением. С другой стороны, интерфейс – это коллекция методов и свойств, которые должны быть реализованы вместе. Следовательно, реализация DOM может определять любые классы, которые считает нужным, но эти классы должны определять методы и свойства различных интерфейсов DOM.

Такая архитектура имеет пару важных следствий. Во-первых, имена классов, используемые в реализации, могут не соответствовать напрямую именам интерфейсов, используемым в стандарте DOM (и в этой книге). Во-вторых, один класс может реализовывать более одного интерфейса. Рассмотрим, например, объект Document. Этот объект является экземпляром некоторого класса, определенного реализацией веб-броузера. Мы не знаем, какой именно это класс, но знаем, что он реализует интерфейс Document; то есть все методы и свойства, определенные интерфейсом Document, доступны нам через объект Document. Поскольку веб-броузеры работают с HTML-документами, мы также знаем, что объект Document реализует интерфейс HTMLDocument и что нам также доступны все методы и свойства, определенные этим интерфейсом. Кроме того, если веб-броузер поддерживает таблицы стилей CSS и реализует модуль CSS DOM, объект Document также реализует интерфейсы DOM DocumentStyle и DocumentCSS. И если веб-броузер поддерживает модули Events и Views, Document также реализует интерфейсы DocumentEvent и DocumentView.

Модель DOM разбита на независимые модули, поэтому в ней определяется несколько дополнительных второстепенных интерфейсов, таких как Docu-

mentStyle, DocumentEvent и DocumentView, каждый из которых определяет только один или два метода. Подобные интерфейсы никогда не реализуются независимо от базового интерфейса Document, и по этой причине я не описываю их отдельно. Ознакомившись с описанием интерфейса Document в «Справочнике по W3C DOM», вы обнаружите, что в нем также перечислены методы и свойства различных дополнительных интерфейсов. Аналогично, посмотрев на описание дополнительных интерфейсов, вы просто найдете перекрестную ссылку на базовый интерфейс, с которым они связаны. Исключения из этого правила составляют случаи, когда дополнительный интерфейс сложен. Например, интерфейс HTMLDocument всегда реализуется тем же объектом, что реализует интерфейс Document, но так как HTMLDocument добавляет значительный объем новой функциональности, я отвел ему отдельную справочную страницу.

Важно также понимать, что поскольку стандарт DOM определяет интерфейсы, а не классы, он не определяет никаких методов-конструкторов. Если, например, требуется создать новый объект Text для вставки в документ, то нельзя просто написать:

```
var t = new Text("это новый текстовый узел");  // Такого конструктора нет!
```

Стандарт DOM не может определять конструкторы, но определяет в интерфейсе Document несколько полезных *методов-фабрик* для создания объектов. Поэтому для создания нового узла Text в документе надо написать:

```
var t = document.createTextNode("это новый текстовый узел");
```

Методы-фабрики, определенные в DOM, имеют имена, которые начинаются со слова «create». Кроме методов-фабрик, определяемых интерфейсом Document, несколько таких методов определяются интерфейсом DOMImplementation и доступны через `document.implementation`.

17.2. Использование базового DOM API

Итак, мы изучили древовидную структуру документа и увидели, как из объектов Node формируется дерево, и теперь можем перейти к более подробному изучению объекта Node и дерева документа. Как отмечалось выше, базовый DOM API не так уж сложен. Следующие разделы содержат примеры, демонстрирующие, как DOM API может применяться для выполнения обычных задач.

17.2.1. Обход документа

Мы уже говорили, что DOM представляет HTML-документ в виде дерева объектов Node. Для любой древовидной структуры наиболее частое выполняемое действие – это обход дерева с поочередным просмотром каждого узла. Один из способов показан в примере 17.1. Это функция на JavaScript, рекурсивно просматривающая узел и все дочерние узлы и подсчитывающая коли-

чество тегов HTML (то есть узлов Element), которые она встретила в процессе обхода. Обратите внимание на свойство childNodes текущего узла. Значением этого свойства является объект NodeList, ведущий себя (в JavaScript) как массив объектов Node. Поэтому функция может перечислить все дочерние узлы данного узла с помощью циклического перебора элементов массива childNodes[]. Функция рекурсивно перечисляет не только все дочерние узлы данного узла, но и все узлы в дереве узлов. Обратите внимание, что эта функция также демонстрирует применение свойства nodeType для определения типа каждого узла.

Пример 17.1. Обход узлов документа

```
<head>
<script>
// Этой функции передается объект DOM Node. Функция проверяет, представляет
// ли этот узел тег HTML, то есть является ли узел объектом Element. Она
// рекурсивно вызывает сама себя для каждого дочернего узла, проверяя их
// таким же образом. Функция возвращает общее число найденных ею объектов
// Element. Если вы вызываете эту функцию, передавая ей объект DOM, она
// выполнит обход всего дерева DOM.
function countTags(n) {                          // n - это Node
    var numtags = 0;                             // Инициализируем счетчик тегов
    if (n.nodeType == 1 /*Node.ELEMENT_NODE*/)   // Проверяем, является ли n
                                                 // объектом
                                                 // Element
        numtags++;                               // Если это так, увеличиваем
                                                 // счетчик
    var children = n.childNodes;                 // Теперь получаем все дочерние
                                                 // элементы n
    for(var i=0; i < children.length; i++) {     // Цикл по всем дочерним элементам
        numtags += countTags(children[i]);       // Рекурсия по всем дочерним
                                                 // элементам
    }
    return numtags;                              // Возвращаем общее число тегов
}
</script>
</head>
<!-- Это пример использования функции countTags() -->
<body onload="alert('В этом документе ' + countTags(document) + ' тегов')">
Это <i>пример</i> документа.
</body>
```

В примере 17.1 надо также обратить внимание, что определенная в нем функция countTags() вызывается из обработчика события onload, поэтому она не будет вызвана, пока документ не загрузится целиком. Это обязательное требование при работе с DOM: нельзя обходить дерево документа или манипулировать с ним до тех пор, пока документ не будет полностью загружен.

В дополнение к свойству childNodes интерфейс Node определяет несколько других полезных свойств. Свойства firstChild и lastChild ссылаются на первый и последний дочерний узел, а nextSibling и previousSibling – на ближай-

шие смежные узлы. (Два узла называются смежными, если имеют один и тот же родительский узел.) Эти свойства предоставляют еще один способ обхода дочерних узлов, который демонстрируется в примере 17.2. Этот пример подсчитывает количество символов во всех узлах Text в теле документа. Обратите внимание, как функция `countCharacters()` использует свойства `firstChild` и `nextSibling` для обхода дочерних узлов.

Пример 17.2. Другой способ обхода документа

```
<head>
<script>
// Этой функции передается объект DOM Node. Функция проверяет, является ли
// этот узел строкой текста, то есть объектом Text. Если
// это так, она возвращает длину строки. Если нет, она рекурсивно вызывает
// себя для каждого дочернего узла и суммирует общую длину найденного
// текста. Обратите внимание, что она перечисляет дочерние узлы с помощью
// свойств firstChild и nextSibling. Заметьте также, что функция не
// выполняет рекурсию, если находит узел Text, так как узлы Text никогда не
// имеют дочерних узлов.
function countCharacters(n) {                // n - это Node
    if (n.nodeType == 3 /*Node.TEXT_NODE*/)  // Проверяем, является ли n
                                             // объектом Text
        return n.length;                     // Если да, возвращаем его длину
    // В противном случае n может иметь дочерние узлы, символы которых
    // нужно подсчитать.
    var numchars = 0;  // Используется для хранения общего числа символов в
                       // дочерних элементах
    // Вместо использования свойства childNodes этот цикл просматривает
    // дочерние узлы узла n с помощью свойств firstChild и nextSibling.
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
        numchars += countCharacters(m);  // Суммирует общее количество
    }                                    // найденных символов
    return numchars;                     // Возвращает общее количество
}                                        // символов
</script>
</head>
<!--
  Обработчик события onload является примером возможного использования
  функции countCharacters(). Обратите внимание, что мы хотим подсчитать
  только символы в теле документа, поэтому мы передаем функции document.body.
-->
<body onload="alert('Длина документа: ' + countCharacters(document.body))">
Это пример документа.<p>Насколько он велик?
</body>
```

17.2.2. Поиск заданного элемента в документе

Возможность обхода всех узлов в дереве документа дает нам средство для поиска определенных узлов. При программировании с использованием DOM API довольно часто возникает задача получения определенного узла из до-

кумента или списка узлов определенного типа. К счастью, DOM API предоставляет функции, облегчающие нам решение этой задачи.

В примере 17.2 мы обращаемся к элементу <body> HTML-документа с помощью выражения JavaScript document.body. Свойство body объекта Document представляет собой удобное специальное свойство и является предпочтительным методом обращения к тегу <body> HTML-документа. Однако при отсутствии такого специального свойства мы могли бы обратиться к тегу <body> следующим образом:

```
document.getElementsByTagName("body")[0]
```

Это выражение вызывает метод getElementsByTagName() и выбирает первый элемент полученного массива. Вызов getElementsByTagName() возвращает массив всех элементов <body> в документе. HTML-документы могут содержать только один тег <body>, поэтому мы знаем, что нас интересует первый элемент полученного массива.[1]

Метод getElementsByTagName() может применяться для получения списка элементов HTML любого типа. Чтобы, например, найти все таблицы в документе, необходимо сделать следующее:

```
var tables = document.getElementsByTagName("table");
alert("Этот документ содержит " + tables.length + " таблиц");
```

Обратите внимание, что поскольку теги HTML не чувствительны к регистру, строки, передаваемые в getElementsByTagName(), также не чувствительны к регистру. То есть предыдущий код находит теги <table>, даже если в коде они представлены как <TABLE>. Метод getElementsByTagName() возвращает элементы в том порядке, в каком они расположены в документе. И наконец, если передать методу getElementsByTagName() специальную строку «*», он вернет список всех элементов в порядке их присутствия в документе. (Этот особый вариант не поддерживается в IE 5 и IE 5.5. См. описание специфического для IE массива Document.all[] в «Справочнике по клиентскому JavaScript».)

Иногда требуется получить не список элементов, а один конкретный элемент документа. Если о структуре документа известно многое, то можно прибегнуть к методу getElementsByTagName(). Так, сделать что-то с четвертым абзацем документа можно при помощи следующего кода:

```
var myParagraph = document.getElementsByTagName("p")[3];
```

Однако, как правило, это не самый лучший (и не самый эффективный) прием, поскольку он очень сильно зависит от структуры документа – вставка нового абзаца в начало документа нарушит работу кода. Вместо этого, когда требуется манипулировать определенными элементами документа, лучше определить для этих элементов атрибут id, задающий уникальное (в преде-

---

[1] Формально в DOM API определено, что getElementsByTagName() возвращает объект NodeList. В привязке DOM к JavaScript объекты NodeList ведут себя как массивы и обычно используются именно таким образом.

лах документа) имя элемента. Тогда элемент можно найти по его идентификатору. Например, отметить специальный абзац документа следующим тегом можно так:

```
<p id="specialParagraph">
```

И тогда найти узел этого абзаца посредством следующего кода JavaScript:

```
var myParagraph = document.getElementById("specialParagraph");
```

Обратите внимание, что метод getElementById() не возвращает массив элементов, как метод getElementsByTagName(). Так как значение каждого атрибута id является (или предполагается) уникальным, getElementById() возвращает только один элемент с соответствующим атрибутом id. Метод getElementById() достаточно важен и довольно часто применяется в программировании DOM.

Оба метода, getElementById() и getElementsByTagName(), относятся к методам объекта Document. Однако объекты Element также определяют метод getElementsByTagName(). Этот метод объекта Element ведет себя так же, как метод объекта Document, за исключением того, что возвращает только элементы, являющиеся потомками элемента, для которого он вызван. Благодаря этому можно, например, сначала использовать getElementById() для поиска определенного элемента, а затем getElementsByTagName() для поиска всех потомков данного типа внутри найденного тега:

```
// Ищет определенный элемент Table внутри документа и подсчитывает
// количество строк в найденной таблице.
var tableOfContents = document.getElementById("TOC");
var rows = tableOfContents.getElementsByTagName("tr");
var numrows = rows.length;
```

И наконец, обратите внимание, что для HTML-документов объект HTMLDocument также определяет метод getElementsByName(). Этот метод похож на getElementById(), но выполняет поиск элементов по атрибуту name, а не по атрибуту id. Кроме того, поскольку атрибут name не обязательно уникален в пределах документа (например, группы переключателей в формах HTML обычно имеют одинаковый атрибут name), getElementsByName() возвращает массив элементов, а не одиночный элемент. Например:

```
// Ищем тег <a name="top">
var link = document.getElementsByName("top")[0];
// Ищем все элементы <input type="radio" name="shippingMethod">
var choices = document.getElementsByName("shippingMethod");
```

17.2.3. Модификация документа

Обход узлов документа может быть полезной функцией, но реальную мощь базового DOM API обеспечивают средства, позволяющие применять JavaScript для динамического изменения документов. Следующие примеры демонстрируют основные приемы модификации документов.

Пример 17.3 включает функцию JavaScript с именем `reverse()`, пример документа и кнопку HTML, которая при щелчке по ней вызывает функцию `reverse()`, передавая ей узел, представляющий элемент `<body>` документа. (Обратите внимание на использование `getElementsByTagName()` в обработчике события кнопки для поиска элемента `<body>`.) Функция `reverse()` выполняет перебор в обратном порядке дочерних узлов переданного в качества аргумента узла и использует методы `removeChild()` и `appendChild()` объекта Node для изменения порядка следования этих элементов на противоположный.

Пример 17.3. Смена порядка узлов документа на противоположный

```
<head><title>Reverse</title>
<script>
function reverse(n) {           // Изменение порядка дочерних узлов узла n
                                // на противоположный
    var kids = n.childNodes;    // Получаем список дочерних элементов
    var numkids = kids.length;  // Выясняем количество дочерних элементов
    for(var i = numkids-1; i >= 0; i--) {  // Выполняем перебор дочерних элементов
                                           // в обратном порядке
        var c = n.removeChild(kids[i]);    // Удаляем дочерний элемент
        n.appendChild(c);                  // Помещаем его в новую позицию
    }
}
</script>
</head>
<body>
<p>paragraph #1<p>paragraph #2<p>paragraph #3 <!-- Пример документа -->
<p>                                <!-- Кнопка для вызова функции reverse()-->
<button onclick="reverse(document.body);"
>Click Me to Reverse</button>
</body>
```

Результат работы примера 17.3, показанный на рис. 17.3, заключается в смене порядка следования абзацев и кнопки на противоположный после щелчка по кнопке.

Стоит отметить пару моментов в примере 17.3. Во-первых, если методу `appendChild()` передается узел, уже являющийся частью документа, он сначала удаляет этот узел, поэтому код можно упростить, опустив вызов `removeChild()`. Во-вторых, имейте в виду, что свойство `childNodes` (как и все объекты NodeList) является «живым»: когда документ изменяется, изменения сразу становятся видимыми благодаря NodeList. Это важная особенность интерфейса NodeList, но она может усложнить написание кода. Например, вызов `removeChild()` изменяет индексы всех смежных узлов, следующих за удаляемым дочерним узлом, поэтому для того чтобы обойти все узлы в NodeList и удалить некоторые из них, писать код цикла следует аккуратно.

Пример 17.4 содержит вариант функции `reverse()` из предыдущего примера. В этом примере используется рекурсия и переставляются не только дочер-

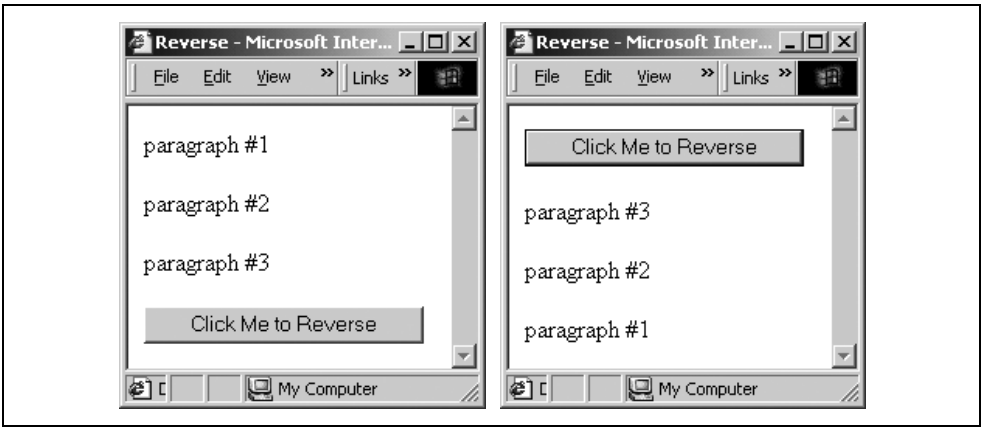

Рис. 17.3. *Документ до и после перестановки элементов*

ние узлы указанного узла, но и все потомки этого узла. Кроме того, когда встречается узел Text, в нем в обратном порядке переставляются символы. Пример 17.4 показывает только код JavaScript для новой функции `reverse()`. Тем не менее он может легко использоваться в HTML-документе, подобном показанному в примере 17.3.

Пример 17.4. Рекурсивная функция обращения порядка узлов

```
// Рекурсивно переставляем в обратном порядке все узлы, разположенные ниже
// узла n, и обращаем узлы Text
function reverse(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {    // Обращаем узлы Text
        var text = n.data;                       // Получаем содержимое узла
        var reversed = "";
        for(var i = text.length-1; i >= 0; i--)  // Обращаем его
            reversed += text.charAt(i);
        n.data = reversed;                       // Сохраняем обращенный текст
    }
    else {  // Для нетекстовых узлов рекурсивно обращаем порядок дочерних
            // узлов
        var kids = n.childNodes;
        var numkids = kids.length;
        for(var i = numkids-1; i >= 0; i--) {       // Цикл по дочерним узлам
            reverse(kids[i]);                       // Рекурсия для обращения
                                                    // дочерних узлов
            n.appendChild(n.removeChild(kids[i]));  // Перемещаем узел в новую
        }                                           // позицию
    }
}
```

Пример 17.4 показывает один из способов изменения текста, отображаемого в документе: просто устанавливается поле `data` соответствующего узла Text. В примере 17.5 показан другой способ. Этот пример определяет функцию `uppercase()`, рекурсивно обходящую дочерние узлы данного узла. Найдя узел

Text, она заменяет его на новый узел Text, содержащий текст исходного узла, преобразованный в верхний регистр. Посмотрите на метод document.createTextNode(), создающий новый текстовый узел, и на метод Node.replaceChild(), заменяющий исходный узел Text на новый. Обратите внимание, что replaceChild() вызывается для родителя заменяемого узла, а не для самого узла. Функция uppercase() использует свойство Node.parentNode для определения родителя заменяемого узла Text.

Кроме определения функции uppercase() пример 17.5 включает два абзаца HTML и кнопку. Когда пользователь щелкает по кнопке, один из абзацев преобразуется в верхний регистр. Каждый абзац идентифицируется уникальным именем, заданным с помощью атрибута id тега <p>. Обработчик события кнопки использует метод getElementById() для получения объекта Element, представляющего требуемый абзац.

Пример 17.5. Замена узлов их эквивалентами в верхнем регистре

```
<script>
// Эта функция рекурсивно обходит узел n и всех его потомков, заменяя
// все узлы Text на их эквиваленты в верхнем регистре.
function uppercase(n) {
    if (n.nodeType == 3 /*Node.TEXT_NODE*/) {
        // Если это узел Text, создаем новый узел Text, содержащий
        // версию текста узла в верхнем регистре, и используем метод
        // replaceChild() родительского узла для замены исходного узла
        // на новый.
        var newNode = document.createTextNode(n.data.toUpperCase());
        var parent = n.parentNode;
        parent.replaceChild(newNode, n);
    }
    else {
        // Если узел не является узлом Text, выполняем цикл по его
        // дочерним узлам и рекурсивно вызываем эту функцию для каждого
        // дочернего узла.
        var kids = n.childNodes;
        for(var i = 0; i < kids.length; i++) uppercase(kids[i]);
    }
}
</script>

<!-- Пример текста. Обратите внимание, что теги <p> имеют атрибуты id. -->
<p id="p1"><i>Это </i> абзац 1.</p>
<p id="p2"><i>Это </i> абзац 2.</p>

<!-- Это кнопка, которая вызывает функцию uppercase(), определенную выше. -->
<!-- Обратите внимание на вызов document.getElementById() для нахождения -->
<!-- требуемого узла.                                                     -->
<button onclick="uppercase(document.getElementById('p1'));">Нажми меня</button>
```

Два предыдущих примера показывают, как изменять содержимое документа путем замены текста, содержащегося в узле Text, и путем замены узла

Text на совершенно новый узел. Также можно добавлять, вставлять, удалять или заменять текст в узле Text с помощью методов appendData(), insertData(), deleteData() и replaceData(). Эти методы не определены непосредственно в интерфейсе Text, а наследуются им от интерфейса CharacterData. Дополнительную информацию о них можно найти в разделе «CharacterData» «Справочника по W3C DOM».

В примерах обхода узлов мы видели, как посредством методов removeChild() и appendChild() переупорядочивать дочерние узлы объекта Node. Заметьте, однако, что при изменении порядка узлов мы не ограничены пределами одного родителя; DOM API позволяет свободно перемещать узлы по дереву документов (но только в пределах одного документа). Пример 17.6 демонстрирует это путем определения функции с именем embolden(), заменяющей указанный узел новым элементом (созданным с помощью метода createElement() объекта Document), представляющим тег HTML <b>, и делающей исходный узел дочерним для нового узла <b>. В HTML-документе из-за этого любой текст в данном узле или в его потомках отображается жирным шрифтом.

Пример 17.6. Изменение родителя узла на элемент <b>

```
<script>
// Эта функция принимает в качестве аргумента узел Node, заменяет его в дереве
// узлом Element, представляющим тег HTML <b>, и затем делает исходный узел
// дочерним для нового элемента <b>.
function embolden(node) {
    var bold = document.createElement("b");  // Создаем новый элемент <b>
    var parent = node.parentNode;            // Получаем родительский узел
    parent.replaceChild(bold, node);         // Заменяем узел тегом <b>
    bold.appendChild(node);                  // Делаем узел дочерним тегу <b>
}
</script>

<!-- Пара простых абзацев -->
<p id="p1"><i>Это </i> абзац #1.</p>
<p id="p2"><i>Это </i> абзац #2.</p>

<!-- Кнопка, вызывающая функцию embolden() для первого абзаца -->
<button onclick="embolden(document.getElementById('p1'));">
Выделить жирным шрифтом</button>
```

Изменять документы можно не только путем вставки, удаления, изменения родителя и другого переупорядочивания узлов, но и простой установкой значений атрибутов элементов документа. Один из возможных способов – применение метода element.setAttribute(). Например:

```
var headline = document.getElementById("headline");  // Поиск элемента
                                                     // с данным именем
headline.setAttribute("align", "center");            // Устанавливаем
                                                     // align='center'
```

В элементах DOM, представляющих атрибуты HTML, определяются свойства JavaScript, соответствующие каждому из их стандартных атрибутов (даже устаревших, таких как align), поэтому получить такой же эффект можно посредством следующего кода:

```
var headline = document.getElementById("headline");
headline.align = "center";  // Устанавливаем атрибут выравнивания.
```

17.2.4. Добавление содержимого в документ

Предыдущие два примера показывают, как перевести содержимое узла Text в верхний регистр и как сделать узел дочерним узлом нового узла <b>. Функция embolden() показала, что можно создавать новые узлы и добавлять их в документ. В документ можно добавить произвольное содержимое, создав узлы Element и Text с помощью методов document.createElement() и document.createTextNode() и добавив их нужным образом в документ. Это демонстрирует пример 17.7, определяющий функцию с именем debug(). Эта функция предоставляет удобный способ вставки отладочных сообщений в прорамму и служит полезной альтернативой встроенной функции alert(). Примерный результат работы этой функции показан на рис. 17.4.

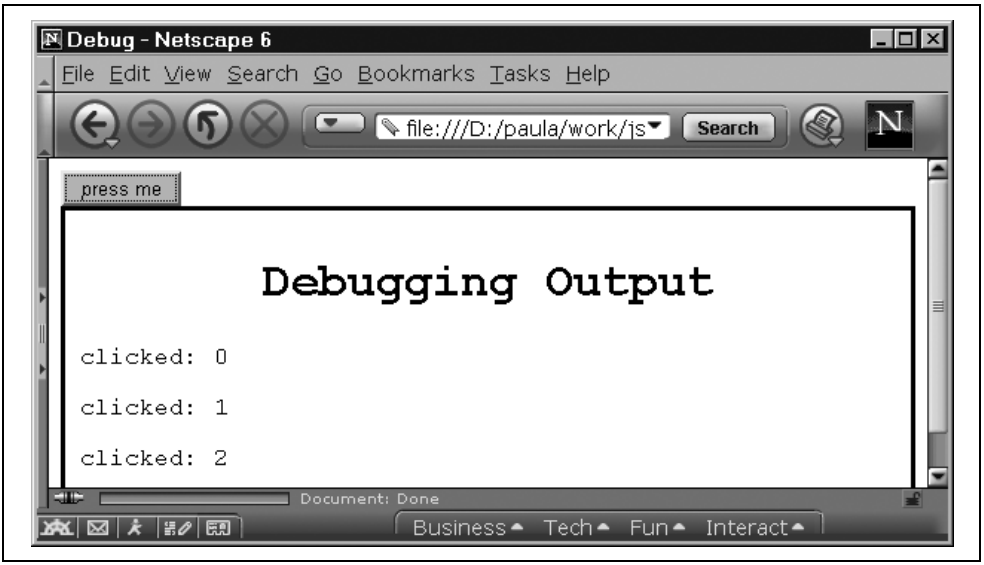

Рис. 17.4. *Результат работы функции debug( )*

При первом вызове debug() функция использует DOM API для создания элемента и вставки его в конец документа. Отладочные сообщения, переданные функции debug() при первом и последующих вызовах, затем вставляются в этот элемент . Они выводятся путем создания узла Text внутри элемента <p> и вставки этого элемента в конец элемента .

Пример 17.7 также демонстрирует удобный, но нестандартный способ добавления нового содержимого в документ. Элемент `<div>`, содержащий отладочные сообщения, выводит большой заголовок по центру страницы. Этот заголовок мог бы быть создан и добавлен в документ так же, как и остальное содержимое, но в данном примере мы вместо этого используем свойство `innerHTML` элемента `<div>`. Установка этого свойства любого элемента равным строке текста HTML приводит к анализу этого кода HTML и вставке его в виде содержимого элемента. Несмотря на то что данное свойство не является частью DOM API, это удобное сокращение, поддерживаемое Internet Explorer 4 и более поздними версиями, а также Netscape 6. Несмотря на свою нестандартность это свойство используется часто и включено в данный пример для полноты изложения.[1]

Пример 17.7. Добавление в документ отладочного вывода

```
/**
 * Эта функция debug выводит текстовые отладочные сообщения в специальную
 * область в конце документа. Это полезная альтернатива использованию alert()
 * для вывода отладочных сообщений.
 **/
function debug(msg) {
    // Если мы еще не создали область, внутри которой должны выводиться наши
    // отладочные сообщения, делаем это. Обратите внимание: чтобы избежать
    // использования еще одной глобальной переменной мы сохраняем узел
    // как свойство этой функции.
    if (!debug.box) {
        // Создаем новый элемент <div>
        debug.box = document.createElement("div");
        // Задаем его внешний вид с помощью стилевых атрибутов CSS
        debug.box.setAttribute("style",
                               "background-color: white; " +
                               "font-family: monospace; " +
                               "border: solid black 3px; " +
                               "padding: 10px;");

        // Добавляем новый элемент <div> в конец документа
        document.body.appendChild(debug.box);

        // Теперь добавляем заголовок к элементу <div>. Обратите внимание, что для
        // анализа фрагмента HTML и вставки его в документ используется свойство
        // innerHTML. Это свойство не является частью стандарта W3C DOM, но
       // поддерживается Netscape 6 и Internet Explorer 4 и более поздними версиями.
        // Мы можем избежать использования innerHTML, явно создав элемент <h1>,
        // установив его стилевой атрибут, присвоив ему узел Text и вставив его
        // в документ, но это свойство упрощает нам задачу.
```

[1] Свойство `innerHTML` особенно полезно, когда требуется вставить в документ большой или сложный кусок текста HTML. Для простых фрагментов HTML методы DOM более эффективны, поскольку не требуют работы анализатора HTML. Обратите внимание, что добавление фрагментов текста к свойству `innerHTML` с помощью оператора `+=` обычно неэффективно.

```
        debug.box.innerHTML = "<h1 style='text-align:center'>Debugging Output</h1>";
    }

    // Начиная с этого места уже debug.box ссылается на элемент
    // <div>, в который мы можем вставить наше отладочное сообщение.
    // Сначала создаем узел <p>, в котором будет находиться сообщение.
    var p = document.createElement("p");
    // Теперь создаем текстовый узел, содержащий сообщение, и добавляем
    // его к <p>
    p.appendChild(document.createTextNode(msg));
    // И добавляем узел <p> к <div>, в котором располагается отладочная выдача
    debug.box.appendChild(p);
}
```

Метод `debug()`, приведенный в примере 17.7, может применяться в HTML-документах, подобных показанному ниже, из которого был сгенерирован экран на рис. 17.4:

```
<script src="Debug.js"></script>  <!-- Включаем функцию debug() -->
<script>var numtimes=0;</script>  <!-- Определяем глобальную переменную -->
<!-- Теперь используем функцию debug() в обработчик события -->
<button onclick="debug(clicked: ' + numtimes++);">press me</button>
```

17.2.5. Работа с фрагментами документа

Базовый DOM API определяет объект DocumentFragment, предоставляющий удобный способ работы с группами узлов документа. DocumentFragment – это особый тип узла, никогда не присутствующий в документе самостоятельно, но служащий в качестве временного контейнера для последовательной коллекции узлов и позволяющий манипулировать этими узлами как одним объектом. Когда DocumentFragment вставляется в документ (с помощью одного из методов объекта Node – `appendChild()`, `insertBefore()` или `replaceChild()`), реально вставляется не DocumentFragment, а все его дочерние узлы.

Для иллюстрации перепишем метод `reverse()` из примера 17.3 с использованием DocumentFragment:

```
function reverse(n) {  // Обращение порядка дочерних элементов узла n
    var f = document.createDocumentFragment();
                      // Создание пустого DocumentFragment
    while(n.lastChild)                // Цикл в обратном порядке по дочерним
          f.appendChild(n.lastChild);  // элементам, перемещающий каждый из
                                       // них в DocumentFragment
    n.appendChild(f);                 // Возвращаем их обратно
                                       // (в новом порядке)
}
```

Создав объект DocumentFragment, его можно использовать, например, с помощью следующего кода:

```
document.getElementsByTagName("p")[0].appendChild(fragment);
```

Обратите внимание, что когда вы вставляете DocumentFragment в документ, его дочерние узлы перемещаются из фрагмента в документ. После вставки фрагмент становится пустым и не может использоваться повторно, пока вы не добавите в него новые дочерние элементы. Мы снова встретимся с объектом DocumentFragment позднее в этой главе, при рассмотрении DOM Range API.

17.2.6. Пример: динамическое создание оглавления

В предыдущих разделах вы видели, как базовый DOM API используется для перебора, изменения и добавления содержимого документа. Пример 17.8, приведенный в конце этого раздела, собирает все эти фрагменты в один более длинный пример. В нем определяется один метод `maketoc()`, который принимает в качестве своего единственного аргумента узел документа. Метод `maketoc()` обходит документ, создает его оглавление и заменяет указанный узел созданным оглавлением. Оглавление создается в результате поиска в документе тегов `<h1>`, `<h2>`, `<h3>`, `<h4>`, `<h5>` и `<h6>`, а также на основе предположения, что эти теги отмечают начало важных разделов документа. Функция `maketoc()` не только создает оглавление, но и вставляет именованные якорные элементы (элементы `<a>` с установленным атрибутом `name` вместо атрибута `href`) перед заголовком каждого раздела, чтобы оглавление могло непосредственно ссылаться на каждый раздел. И наконец, `maketoc()` также вставляет в начало каждого раздела ссылки обратно на оглавление. Когда читатель дойдет до нового раздела, он сможет либо прочитать этот раздел, либо вернуться по ссылке обратно к оглавлению и выбрать другой раздел. Оглавление, генерируемое функцией `maketoc()`, показано на рис. 17.5.

Функция `maketoc()` может представлять интерес для тех, кто сопровождает и редактирует длинные документы, разбитые на разделы с помощью `<h1>`, `<h2>` и подобных им тегов. Оглавления очень полезны в длинных документах, но если документ редактируется часто, то бывает сложно обеспечивать синхронизацию оглавления с самим документом. Оглавление для этой книги создавалось автоматически путем обработки текста книги. Метод `maketoc()` позволяет делать нечто подобное с веб-документами. Эта функция может применяться к HTML-документам, подобным приведенному ниже:

```
<script src="TOC.js"></script>  <!-- Загружаем функцию maketoc() -->
<!-- Вызываем функцию maketoc(), когда документ загружен полностью -->
<body onload="maketoc(document.getElementById('placeholder'))">
<!-- Этот элемент span будет заменен сгенерированным оглавлением -->
<span id="placeholder">Оглавление</span>
// ... оставшаяся часть документа ...
```

Другой способ применения функции `maketoc()` заключается в создании оглавления по просьбе читателя. Для этого надо добавить в документ ссылку (или кнопку), которая при щелчке заменяется сгенерированным оглавлением:

```
<a href="#" onclick="maketoc(this); return false;">Показать оглавление</a>
```

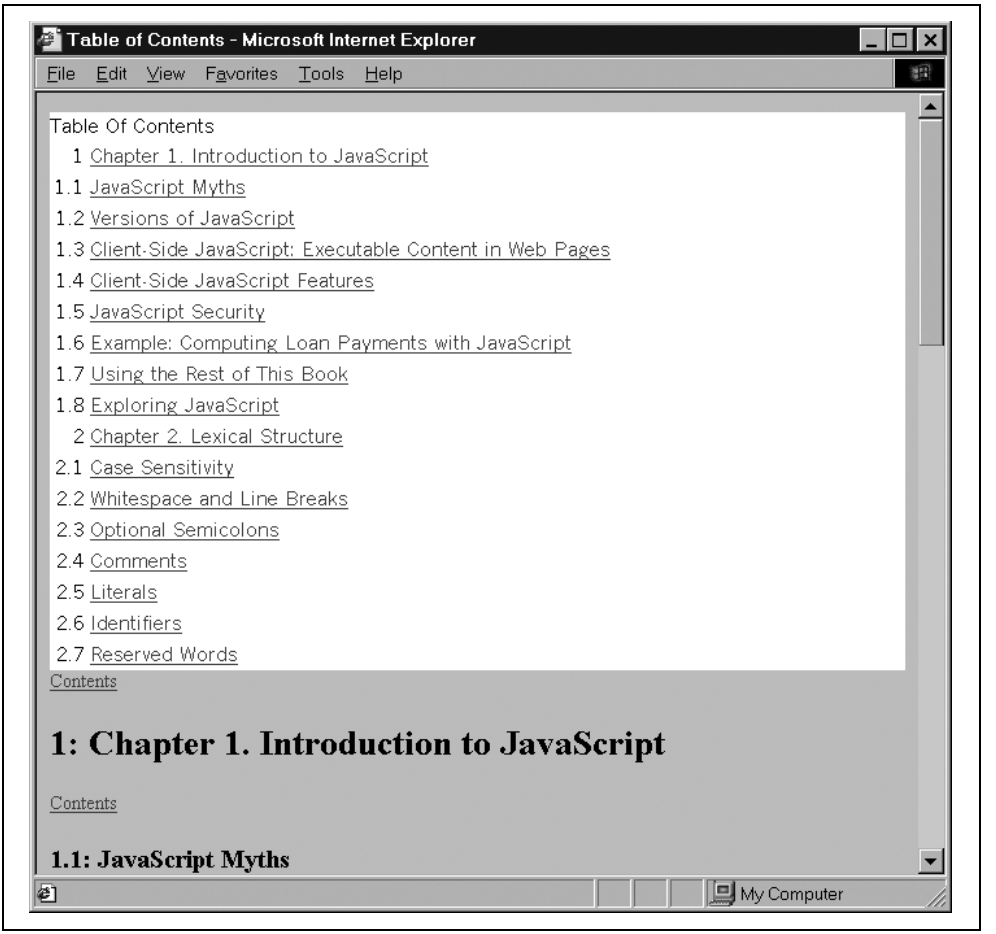

Рис. 17.5. *Динамически созданное оглавление*

Далее следует код функции `maketoc()`. Пример 17.8 длинен, но хорошо комментирован и основан на приемах, которые уже демонстрировались. Он заслуживает изучения как демонстрация мощи DOM API. Обратите внимание, что функция `maketoc()` обращается к двум вспомогательным функциям. Для сохранения модульности эти функции определены внутри самой `maketoc()`. Это позволяет избежать добавления дополнительных ненужных функций в глобальное пространство имен.

Пример 17.8. Автоматическая генерация оглавления

```
/**
 * Создание оглавления для текущего документа и вставка его в документ путем
 * замены узла, указанного аргументом replace.
 **/
function maketoc(replace) {
```

```
// Создаем элемент <div>, являющийся корнем дерева оглавления.
var toc = document.createElement("div");

// Устанавливаем цвет фона и текста для оглавления.
// О свойстве style мы узнаем в следующей главе.
toc.style.backgroundColor = "white";
toc.style.fontFamily = "sans-serif";

// Начинаем оглавление с якорного элемента, на который мы сможем ссылаться.
var anchor = document.createElement("a");  // Создаем узел <a>
anchor.setAttribute("name", "TOC");        // Даем ему имя
toc.appendChild(anchor);                   // Вставляем

// Телом якорного элемента делаем заголовок оглавления
anchor.appendChild(document.createTextNode("Оглавление"));

// Создаем элемент <table>, в котором будет находиться оглавление,
// и добавляем его.
var table = document.createElement("table");
toc.appendChild(table);

// Создаем элемент <tbody>, в котором находятся строки оглавления.
var tbody = document.createElement("tbody");
table.appendChild(tbody);

// Инициализируем массив, в котором будут отслеживаться номера разделов.
var sectionNumbers = [0,0,0,0,0,0];

// Рекурсивно обходим тело документа, выбирая разделы, отмеченные тегами
// <h1>, <h2>, ..., и используем их для создания оглавления, добавляя
// строки в таблицу.
addSections(document.body, tbody, sectionNumbers);

// Наконец, вставляем оглавление в документ, заменяя узел, указанный
// аргументом replace, на поддерево оглавления.
replace.parentNode.replaceChild(toc, replace);

// Этот метод рекурсивно обходит дерево, корнем которого является узел n,
// выбирая теги <h1> - <h6>, и использует содержимое этих тегов для построения
// оглавления, добавляя строки в таблицу HTML, указанную аргументом toc.
// Он использует массив sectionNumbers для отслеживания текущего номера раздела.
// Чтобы не быть видимой снаружи, эта функция определяется внутри maketoc().
// Единственная функция, экспортируемая этим модулем JavaScript, - это maketoc().
function addSections(n, toc, sectionNumbers) {
    // Цикл по всем дочерним узлам n.
    for(var m = n.firstChild; m != null; m = m.nextSibling) {
       // Проверяем, является ли m элементом-заголовком. Было бы хорошо, если бы
        // мы могли просто использовать его (m instanceof HTMLHeadingElement),
        // но это не обязательно по спецификации и не работает в IE.
        // Следовательно, мы должны проверить имя тега на его равенство H1-H6.
        if ((m.nodeType == 1) &&  /* Node.ELEMENT_NODE */
            (m.tagName.length == 2) && (m.tagName.charAt(0) == "H")) {
            // Выясняем уровень данного заголовка.
            var level = parseInt(m.tagName.charAt(1));
            if (!isNaN(level) && (level >= 1) && (level <= 6)) {
                // Увеличиваем номер раздела для этого уровня.
```

```
    sectionNumbers[level-1]++;
    // И сбрасываем все номера заголовков меньших уровней в 0.
    for(var i = level; i < 6; i++) sectionNumbers[i] = 0;
    // Теперь объединяем номера разделов для всех уровней, чтобы
    // получить номер раздела в виде "2.3.1".
    var sectionNumber = "";
    for(var i = 0; i < level; i++) {
        sectionNumber += sectionNumbers[i];
        if (i < level-1) sectionNumber += ".";
    }

  // Создаем якорный элемент, чтобы отметить начало данного раздела.
    // Это будет цель ссылки, которую мы добавляем в оглавление.
    var anchor = document.createElement("a");
    anchor.setAttribute("name", "SECT"+sectionNumber);

    // Создаем ссылку обратно на оглавление и делаем ее дочерним
    // узлом якоря.
    var backlink = document.createElement("a");
    backlink.setAttribute("href", "#TOC");
    backlink.appendChild(document.createTextNode("Contents"));
    anchor.appendChild(backlink);

    // Вставляем якорь в документ непосредственно перед
    // заголовком раздела.
    n.insertBefore(anchor, m);

    // Теперь создаем ссылку на этот раздел. Ниже она будет
    // добавлена в оглавление.
    var link = document.createElement("a");
    link.setAttribute("href", "#SECT" + sectionNumber);
  // Получаем текст заголовка с помощью функции, определяемой ниже.
    var sectionTitle = getTextContent(m);
    // Используем текст заголовка в качестве содержимого ссылки.
    link.appendChild(document.createTextNode(sectionTitle));

    // Создаем новую строку для оглавления.
    var row = document.createElement("tr");
    // Создаем два столбца для строки.
    var col1 = document.createElement("td");
    var col2 = document.createElement("td");
    // Делаем первый столбец выровненным по правому краю и
    // помещаем в него номер раздела.
    col1.setAttribute("align", "right");
    col1.appendChild(document.createTextNode(sectionNumber));
    // Во второй столбец помещаем ссылку на раздел.
    col2.appendChild(link);
    // Добаляем столбцы в строку, а строку в таблицу.
    row.appendChild(col1);
    row.appendChild(col2);
    toc.appendChild(row);

    // Изменяем сам элемент заголовка раздела, чтобы добавить
    // номер раздела в качестве части заголовка.
```

```
                    m.insertBefore(document.createTextNode(sectionNumber+": "),
                                   m.firstChild);
                }
            }
            else {  // В противном случае это не элемент заголовка, поэтому выполняем
                addSections(m, toc, sectionNumbers);                    //рекурсию
            }
        }
    }

    // Эта вспомогательная функция обходит узел n, возвращая содержимое всех
    // найденных узлов Text и отбрасывая все теги HTML. Эта функция также определяется
    // как вложенная, поэтому она является закрытой функцией данного модуля.
    Function getTextContent(n) {
        var s = '';
        var children = n.childNodes;
        for(var i = 0; i < children.length; i++) {
            var child = children[i];
            if (child.nodeType == 3 /*Node.TEXT_NODE*/) s += child.data;
            else s += getTextContent(child);
        }
        return s;
    }
}
```

17.2.7. Работа с XML-документами

Веб-броузеры отображают документы HTML, но все большее значение как источники данных приобретают документы XML. DOM позволяет выполнять обход и манипуляции как HTML-, так и XML-документами, поэтому методы DOM могут применяться для загрузки документа XML, извлечения из него информации и динамического создания HTML-версии этой информации для отображения в веб-броузере. В примере 17.9 показано, как это можно реализовать в Netscape 6.1 и Internet Explorer 6. Это файл HTML, состоящий в основном из кода JavaScript. Предполагается, что он загружен со строки запроса, содержащей относительный URL загружаемого файла данных. Например, этот файл примера можно вызвать с помощью следующего URL:

```
file://C:/javascript/DisplayEmployeeData.html?data.xml
```

DisplayEmployeeData.html – это имя файла примера, а *data.xml* – имя используемого им файла XML. Файл XML должен содержать данные, отформатированные следующим образом:

```
<employees>
  <employee name="J. Doe"><job>Programmer</job><salary>32768</salary></employee>
  <employee name="A. Baker"><job>Sales</job><salary>70000</salary></employee>
  <employee name="Big Cheese"><job>CEO</job><salary>1000000</salary></employee>
</employees>
```

Пример содержит две функции JavaScript. Первая, `loadXML()`, представляет собой универсальную функцию, способную загрузить любой файл XML. Она содержит стандартный код DOM Level 2 для загрузки документа XML, а также код, использующий API от Microsoft для выполнения того же действия. Новое в этом примере – лишь создание нового объекта Document с помощью метода `DOMImplementation.createDocument()` и вызов метода `load()` этого объекта. Здесь важно отметить, что документы не загружаются мгновенно, поэтому вызов `loadXML()` возвращает управление до окончания загрузки документа. Поэтому мы передаем в `loadXML()` ссылку на другую функцию, которая должна быть вызвана, когда загрузка документа завершится.

Другая функция в примере – `makeTable()`. Это функция, передаваемая в качестве параметра функции `loadXML()`. Когда загрузка файла XML завершится, `loadXML()` передает объект Document, представляющий файл XML, и URL файла в функцию `makeTable()`. Функция `makeTable()` использует рассмотренные нами ранее методы DOM для извлечения информации из документа XML и вставки ее в таблицу в HTML-документе, отображаемом броузером. Эта функция также иллюстрирует применение некоторых методов работы с таблицами, определенных в HTMLTableElement, HTMLTableRowElement и связанных с ними интерфейсах. Полные описания этих интерфейсов и их методов приведены в «Справочнике по W3C DOM». Методы и свойства DOM, задействованные в этой функции, просты, но они применяются здесь в сложных комбинациях. Тщательно изучите код этого примера, и вы его поймете.

Пример 17.9. Загрузка и чтение данных из XML-документа

```
<head><title>Employee Data</title>
<script>
// Эта функция загружает документ XML с указанного URL и, когда документ
// будет полностью загружен, передает этот документ и его URL указанной
// функции-обработчику. Эта функция работает с любым документом XML.
function loadXML(url, handler) {
    // Если поддерживается DOM Level 2, используем стандартную методику
    if (document.implementation && document.implementation.createDocument) {
        // Создаем новый объект Document
        var xmldoc = document.implementation.createDocument("", "", null);
        // Указываем, что должно произойти, когда загрузка будет завершена
        xmldoc.onload = function() { handler(xmldoc, url); }
        // И сообщаем, какой URL нужно загрузить
        xmldoc.load(url);
    }
    // В противном случае используем фирменный API от Microsoft
    // для Internet Explorer
    else if (window.ActiveXObject) {
        var xmldoc = new ActiveXObject("Microsoft.XMLDOM");   // Создаем xmldoc
        xmldoc.onreadystatechange = function() {              // Задаем onload
            if (xmldoc.readyState == 4) handler(xmldoc, url);
        }
        xmldoc.load(url);                                     //Начинаем загрузку!
    }
}
```

```
// Эта функция формирует таблицу HTML из данных, прочитанных из переданного ей
// документа XML.
function makeTable(xmldoc, url) {
    // Создаем объект <table> и вставляем его в документ.
    var table = document.createElement("table");
    table.setAttribute("border", "1");
    document.body.appendChild(table);

    // Используем методы HTMLTableElement и родственных ему интерфейсов
    // для определения шапки таблицы и заголовка, задающего имя каждого столбца.
    var caption = "Данные о работниках из файла " + url;
    table.createCaption().appendChild(document.createTextNode(caption));
    var header = table.createTHead();
    var headerrow = header.insertRow(0);
    headerrow.insertCell(0).appendChild(document.createTextNode("Name"));
    headerrow.insertCell(1).appendChild(document.createTextNode("Job"));
    headerrow.insertCell(2).appendChild(document.createTextNode("Salary"));

    // Теперь находим элементы <employee> в нашем документе xmldoc.
    var employees = xmldoc.getElementsByTagName("employee");

    // Цикл по всем элементам <employee>.
    for(var i = 0; i < employees.length; i++) {
        // Для каждого работника извлекаем его имя, должность и зарплату,
        // используя стандартные методы DOM. Имя извлекается из атрибута.
        // Другие значения хранятся в текстовых узлах в тегах <job> и <salary>.
        var e = employees[i];
        var name = e.getAttribute("name");
        var job = e.getElementsByTagName("job")[0].firstChild.data;
        var salary = e.getElementsByTagName("salary")[0].firstChild.data;

        // Теперь, когда у нас есть данные о работниках, используем методы
        // таблицы для создания новой строки, а затем методы строки для
        // создания новых ячеек, содержащих данные в виде узлов Text.
        var row = table.insertRow(i+1);
        row.insertCell(0).appendChild(document.createTextNode(name));
        row.insertCell(1).appendChild(document.createTextNode(job));
        row.insertCell(2).appendChild(document.createTextNode(salary));
    }
}
</script>
</head>
<!--
Тело документа не содержит статического текста; все генерируется динамически
функцией makeTable(). Обработчик события onload начинает свою работу с вызова
функции loadXML() для загрузки файла данных в формате XML. Обратите внимание
на использование location.search для кодирования имени XML-файла в строке
запроса. Загружайте этот файл HTML с помощью URL вроде
DisplayEmployeeData.html?data.xml.
-->
<body onload="loadXML(location.search.substring(1), makeTable)">
</body>
```

17.3. Совместимость DOM с Internet Explorer 4

IE 4 не совместим с DOM, но в нем есть средства, сходные с базовым DOM API. Эти средства не являются частью стандарта DOM и не совместимы с Netscape, но совместимы с более поздними версиями IE. Здесь приведена сводка этих возможностей, а подробности можно найти в «Справочнике по клиентскому JavaScript» в этой книге.

17.3.1. Обход документа

Стандарт DOM указывает, что во всех объектах Node, включая объект Document и все объекты Element, имеется массив `childNodes[]`, содержащий дочерние узлы данного узла. IE 4 не поддерживает `childNodes[]`, но предоставляет очень похожий на него массив `children[]` в объектах Document и HTMLElement. Поэтому легко написать рекурсивную функцию, аналогичную показанной в примере 17.1, для обхода всех элементов HTML в документе IE 4.

Однако между массивом `children[]` в IE 4 и стандартным массивом `childNodes[]` в DOM есть одно существенное различие. В IE 4 нет типа узла Text, и в нем строки текста не рассматриваются как дочерние узлы. Следовательно, тег `<p>`, содержащий только обычный текст без разметки, в IE 4 имеет пустой массив `children[]`. Однако, как мы скоро увидим, текстовое содержимое тега `<p>` в IE 4 доступно через свойство `innerText`.

17.3.2. Поиск элементов документа

IE 4 не поддерживает методы `getElementById()` и `getElementsByTagName()` объекта Document. Вместо этого у объекта Document и у всех элементов документа есть свойство-массив с именем `all[]`. Как следует из имени, этот массив представляет *все* элементы документа или все элементы, содержащиеся в другом элементе. Обратите внимание, что `all[]` не просто представляет дочерние узлы документа или элемента – он содержит всех потомков, независимо от глубины вложенности.

Массив `all[]` может использоваться несколькими способами. Если он индексируется с помощью целого индекса n, то возвращает $n+1$-й элемент документа или родительского элемента. Например:

```
var e1 = document.all[0];  // Первый элемент документа
var e2 = e1.all[4];        // Пятый элемент в элементе 1
```

Элементы нумеруются в порядке их расположения в исходном тексте документа. Обратите внимание на одно существенное различие между API IE 4 и стандартом DOM: в IE нет понятия текстовых узлов, поэтому массив `all[]` содержит только элементы документа, но не текст внутри них.

Обычно значительно полезнее иметь возможность ссылаться на элементы документа по имени, чем по номеру. Эквивалентом `getElementById()` в IE 4 является индексирование массива `all[]` с помощью строки, а не числа. Когда

вы пользуетесь этой возможностью, IE 4 возвращает элемент, у которого атрибут id или name равен указанному значению. Если имеется более одного такого элемента (что возможно, так как часто имеется несколько элементов формы, например переключателей, с одинаковыми атрибутами name), результатом будет массив этих элементов. Например:

```
var specialParagraph = document.all["special"];
var radioboxes = form.all["shippingMethod"];  // Может вернуть массив
```

JavaScript также позволяет записывать эти выражения, указывая индекс массива как имя свойства:

```
var specialParagraph = document.all.special;
var radioboxes = form.all.shippingMethod;
```

Подобное применение массива all[] предоставляет ту же базовую функциональность, что и методы getElementById() и getElementsByName(). Основное отличие состоит в том, что массив all[] объединяет возможности этих двух методов, что может привести к проблемам при непреднамеренном использовании одинаковых значений атрибутов id и name для несвязанных элементов.

У массива all[] есть необычная особенность: наличие метода tags(), который может применяться для получения массива элементов по имени тега:

```
var lists = document.all.tags("UL"); // Ищет все теги <ul> в документе
var items = lists[0].all.tags("LI"); // Ищет все теги <li> внутри первого <ul>
```

Этот синтаксис IE 4 предоставляет практически ту же функциональность, что и метод getElementsByTagName() объектов Document и Element в DOM. Обратите внимание, что в IE 4 имя тега должно содержать только строчные буквы.

17.3.3. Модификация документов

Как и стандарт DOM, IE 4 предоставляет доступ к атрибутам тегов HTML как к свойствам соответствующих объектов HTMLElement. Поэтому можно изменять документ, отображаемый в IE 4, путем динамического изменения атрибутов HTML. Если модификация атрибута приводит к изменению размера какого-либо элемента, документ переформатируется для согласования с новым размером элемента. Объект HTMLElement в IE 4 также определяет методы setAttribute(), getAttribute() и removeAttribute(). Они аналогичны методам с теми же именами, определенными в объекте Element стандартного DOM API.

Стандарт DOM определяет API, позволяющий создавать новые узлы, вставлять узлы в дерево документа, менять родителей для узлов и перемещать узлы внутри дерева. IE 4 не может этого делать. Однако вместо этого во всех объектах HTMLElement в IE 4 определено свойство innerHTML. Установка этого свойства равным строке текста HTML позволяет заменить содержимое элемента чем угодно. Поскольку свойство innerHTML представляет собой столь

мощное средство, оно также было реализовано в броузере Netscape 6 (и в Mozilla, от которого он является производным), даже несмотря на то, что это свойство не является частью стандарта DOM. Применение `innerHTML` было продемонстрировано в примере 17.7.

IE 4 также определяет несколько сходных свойств и методов. Свойство `outerHTML` заменяет содержимое элемента и целиком сам элемент на указанную строку HTML. Свойства `innerText` и `outerText` аналогичны `innerHTML` и `outerHTML` за исключением того, что рассматривают строку как обычный текст и не анализируют ее как HTML. И наконец, методы `insertAdjacentHTML()` и `insertAdjacentText()` не затрагивают сам элемент, но вставляют новое HTML- или текстовое содержимое рядом (до или после, внутри или снаружи) с элементом. Эти свойства и функции не настолько часто используются, как `innerHTML` и не были реализованы в Netscape 6. Подробные сведения можно найти на странице «HTMLElement» в «Справочнике по клиентскому JavaScript».

17.4. Совместимость DOM с Netscape 4

Netscape 4 даже близко не подходит к реализации стандарта DOM. В частности, Netscape 4 не предоставляет никакой возможности обращаться к произвольным атрибутам документов и устанавливать их. Конечно, Netscape 4 поддерживает DOM API Level 0, поэтому такие элементы, как формы и ссылки, доступны через массивы `forms[]` и `links[]`, но общий способ обхода дочерних узлов этих элементов или установки в них произвольных атрибутов отсутствует. Кроме того, Netscape 4 не позволяет переформатировать содержимое документа в ответ на изменение размера элемента.

Несмотря на эти ограничения Netscape 4 предоставляет API, обеспечивающий доступ и манипуляцию ключевыми «динамическими элементами», используемыми для реализации эффектов DHTML. В Netscape 4 эти элементы известны как *слои* (*layers*); они «плавают» над остальной частью документа и могут перемещаться, изменять размер и другие свои параметры независимо от других элементов документа. Слои обычно реализуются с помощью таблиц стилей CSS; Netscape 4 Layer API подробно обсуждается в главе 18.

Далее поясняется, как создавать и изменять содержимое отдельных элементов-слоев в документе. Netscape 4 не поддерживает ничего похожего на стандарт DOM, но его Layer API позволяет добиваться некоторых динамических эффектов, возможных при работе со стандартным API. Обратите внимание, что Layer API был представлен в W3C на рассмотрение в качестве части стандарта DOM, но никакая из частей этого API никогда не была стандартизована. Поскольку Netscape 6 базируется на полностью переписанном коде Netscape 4, Layer API был исключен и не поддерживается в Netscape 6 (и в Mozilla).

Слои могут создаваться в документе с помощью тега `<layer>`, фирменного расширения HTML компании Netscape. Однако чаще всего слои создаются в документе Netscape 4 с помощью стандартных атрибутов позиционирования

CSS (которые будут подробно описаны в главе 18). Любой элемент, сделанный динамическим с помощью атрибутов стиля CSS, рассматривается Netscape 4 как слой, и с ним можно манипулировать посредством Layer API. (Однако обратите внимание, что Netscape 4 не позволяет делать все элементы динамическими. Поэтому обычно вокруг любого элемента, который должен быть динамическим, используется элемент-обертка `<div>`.) JavaScript также может динамически создавать слои с помощью конструктора `Layer()`, о котором рассказано в «Справочнике по клиентскому JavaScript» в этой книге.

После того как вы создали в документе динамические элементы, или слои, Netscape 4 позволяет вам обращаться к ним с помощью простого расширения DOM API Level 0. Вы можете обращаться к слоям с помощью массива `layers[]` объекта Document так же, как вы обращаетесь к элементам формы с помощью массива `forms[]` и к элементам-изображениям с помощью массива `images[]`. Если первый слой в документе имеет атрибут `name`, равный «layer1», то можно ссылаться на этот элемент-слой любым из следующих выражений:

```
document.layers[0]          // Индексирование массива с помощью числа
document.layers['layer1']   // Индексирование массива с помощью
                            // имени элемента
document.layer1             // Именованные слои становятся свойствами документа
```

Если у слоя нет атрибута `name`, но есть атрибут `id`, в качестве имени слоя выступает значение этого атрибута.

Слои документов представлены объектами Layer, определяющими несколько полезных свойств и методов, которые можно использовать для перемещения, изменения размера, скрытия и установки места слоя в стеке. Эти свойства и методы связаны с атрибутами стилей CSS и будут обсуждаться в главе 18. Самая интересная особенность объекта Layer состоит в том, что в нем имеется собственный объект Document: содержимое слоя рассматривается как документ, совершенно отдельный от документа, содержащего слой. Это позволяет изменять отображаемое слоем содержимое, динамически переписывая его с помощью методов `document.write()` и `document.close()`. Также можете динамически загружать документы в слой с помощью метода `Layer.load()`. И наконец, заметим, что слои сами могут содержать слои, и на такие вложенные слои можно ссылаться с помощью примерно таких выражений:

```
// Второй слой, вложенный в слой по имени "mylayer"
document.mylayer.document.layers[1];
```

17.5. Дополнительные методы: Traversal API и Range API

В предыдущих разделах этой главы мы обсуждали базовый DOM API, который предоставляет базовые методы, позволяющие обходить документ и манипулировать им. Стандарт DOM также определяет другие дополнительные модули API, наиболее важные из которых мы обсудим в следующих главах. Два дополнительных модуля в основном предназначены для удобства и работают поверх базового API. Traversal API определяет развитые технологии для обхода документа и фильтрации узлов, не представляющих интереса. Range API определяет методы, позволяющие манипулировать непрерывными областями содержимого документа, даже когда это содержимое не начинается или не заканчивается на границе узла. Traversal API и Range API кратко представлены в следующих разделах. Полное описание приведено в «Справочнике по W3C DOM». Range API реализован в Netscape 6.1 (и частично в Netscape 6), а полная поддержка Traversal API ожидается в Mozilla 1.0, значит, будущий релиз Netscape тоже будет его поддерживать. На момент написания этой книги IE не поддерживал ни один из этих API.

17.5.1. DOM Traversal API

В начале этой главы мы видели технологию обхода дерева документа путем рекурсивного просмотра всех узлов по очереди. Эта технология полезна, но избыточна – обычно обходить все узлы документа не требуется. Но нам может потребоваться просмотреть в документе только элементы `<img>` или обойти только поддеревья элементов `<table>`. Traversal API предоставляет развитые технологии для такого выборочного обхода документа. Как отмечалось выше, Traversal API не обязателен и на момент написания этой книги не был реализован в основных броузерах шестого поколения. Проверить, поддерживается ли этот API совместимым с DOM броузером, позволяет, например, следующий код:

```
document.implementation.hasFeature("Traversal", 2.0)
                                    // True, если поддерживается
```

17.5.1.1. NodeIterator и TreeWalker

Traversal API содержит два ключевых объекта, предоставляющих различное фильтрованное представление документа. Объект NodeIterator обеспечивает «плоское» последовательное представление узлов документа и поддерживает фильтрацию. Можно определить объект NodeIterator, отфильтровывающий все узлы документа, кроме тегов `<img>`, и представляющий элементы изображений в виде списка. Методы `nextNode()` и `prevousNode()` объекта NodeIterator позволяют двигаться вперед и назад по списку. Обратите внимание, что NodeIterator позволяет обходить выборочные части документа без рекурсии – можно просто использовать NodeIterator в цикле, повторно вызывая `nextNode()` до тех пор, пока не отыщутся нужные узел или

узлы или пока nextNode() не вернет null, указывая, что достигнут конец документа.

Другой ключевой объект Traversal API – это TreeWalker. Этот объект также предоставляет фильтрованное представление документа и позволяет обходить фильтрованный документ с помощью методов nextNode() и previousNode(), но не делает дерево документа плоским. TreeWalker сохраняет древовидную структуру документа (хотя она может быть сильно изменена из-за фильтрации узлов) и позволяет перемещаться по дереву с помощью методов firstChild(), lastChild(), nextSibling(), previousSibling() и parentNode(). Можно использовать TreeWalker вместо NodeIterator, когда требуется самостоятельно обойти дерево, а не просто осуществить цикл с помощью nextNode(), или когда надо выполнить более сложный обход, например, пропуская некоторые поддеревья.

Объект Document определяет методы createNodeIterator() и createTreeWalker() для создания объектов NodeIterator и TreeWalker. Практический способ, позволяющий проверить, поддерживает ли броузер Traversal API, состоит в проверке существования этих методов:

```
if (document.createNodeIterator && document.createTreeWalker) {
    /* Можно использовать Traversal API */
}
```

Методам createNodeIterator() и createTreeWalker() передаются четыре одинаковых аргумента, и отличие состоит только в типе возвращаемых ими объектов. Первый аргумент – это узел, с которого должен начаться обход. Это может быть объект Document, если надо обойти весь документ или осуществить цикл по нему, или другой узел, если надо обойти только поддерево документа. Второй аргумент – это число, указывающее тип узлов, который должен возвращать NodeIterator или TreeWalker. Этот аргумент получается путем вычисления суммы одной или более констант SHOW, определяемых объектом NodeFilter (обсуждается в следующем разделе). Третий аргумент обоих методов – это необязательная функция, предназначенная для задания более сложного фильтра, чем просто включение или пропуск узлов в зависимости от их типа (тоже описана в следующем разделе). Последний аргумент – это логическое значение, указывающее, должны ли узлы ссылок на сущности в документе разворачиваться в процессе обхода. Это может быть полезно при работе с документами XML, а веб-программисты, работающие с документами HTML, могут игнорировать ее и передавать false.

17.5.1.2. Фильтрация

Одной из наиболее важных характеристик NodeIterator и TreeWalker является их избирательность, то есть возможность отфильтровывать узлы, которыми вы не хотите заниматься. Как говорилось выше, с помощью второго и (необязательно) третьего аргументов методов createNodeIterator() и createTreeWalker() вы определяете узлы, которые вас интересуют. Эти аргументы задают два уровня фильтрации. Первый уровень просто принимает или про-

пускает узлы в зависимости от их типа. Объект NodeFilter определяет числовые константы для каждого типа узлов, а типы узлов задаются сложением (или при помощи поразрядного оператора ИЛИ (|)) необходимых констант.

Например, если вас интересуют только узлы Element и Text, то в качестве второго аргумента может выступать следующее выражение:

```
NodeFilter.SHOW_ELEMENT + NodeFilter.SHOW_TEXT
```

А если только узлы Element, то

```
NodeFilter.SHOW_ELEMENT
```

Если же вас интересуют все узлы либо вы не хотите пропускать какие-либо узлы только на основе их типа, используйте специальную константу

```
NodeFilter.SHOW_ALL
```

А если все типы узлов, кроме комментариев, используйте

```
~NodeFilter.SHOW_COMMENT
```

(Если вы забыли смысл оператора ~, обратитесь к главе 5.) Обратите внимание, что этот первый уровень фильтрации применяется к отдельным узлам, но не к их дочерним элементам. Если вторым аргументом является `NodeFilter.SHOW_TEXT`, ваш NodeIterator или TreeWalker не возвращает узлов элементов, но и не отбрасывает их полностью; он все равно обходит поддерево ниже узлов Element в поиске интересующих вас узлов Text.

Любые узлы, прошедшие эту фильтрацию по типу, могут быть пропущены через второй уровень фильтрации. Этот второй фильтр реализуется функцией, которую определяете вы и которая, следовательно, может выполнять сколь угодно сложную фильтрацию. Если этот вид фильтрации не нужен, можно указать `null` в качестве значения третьего аргумента `createNodeIterator()` или `createTreeWalker()`. Но если такая фильтрация все-таки требуется, то надо передать в качестве третьего аргумента функцию.

Функция должна принимать один аргумент-узел, оценивать этот узел и возвращать значение, указывающее, должен ли этот узел отфильтровываться. Может возвращаться одно из трех значений, определенных тремя константами `NodeFilter`. Если функция-фильтр возвращает `NodeFilter.FILTER_ACCEPT`, то узел возвращается объектом NodeIterator или TreeWalker. Если функция возвращает `NodeFilter.FILTER_SKIP`, то узел отфильтровывается и не возвращается объектом NodeIterator или TreeWalker. Однако дочерние узлы этого узла все еще участвуют в обходе. Если вы работаете с TreeWalker, функция может также возвращать значение `NodeFilter.FILTER_REJECT`, указывающее, что узел не должен возвращаться и даже не должен участвовать в обходе.

В примере 17.10 демонстрируется создание и использование NodeIterator, и он должен сделать предыдущее описание понятным. Заметьте, однако, что на момент написания этой книги ни один из основных веб-броузеров не поддерживал Traversal API, поэтому данный пример не тестировался!

Пример 17.10. Создание и использование NodeIterator

```
// Определяет функцию-фильтр, принимающую только элементы <img>
function imgfilter(n) {
    if (n.tagName == 'IMG') return NodeFilter.FILTER_ACCEPT;
    else return NodeFilter.FILTER_SKIP;
}

// Создание NodeIterator для поиска тегов <img>
var images = document.createNodeIterator(document,  // Обход всего документа
    /* Поиск только узлов Element */       NodeFilter.SHOW_ELEMENT,
    /* Отфильтровываем все, кроме <img> */ imgfilter,
    /* В HTML-документах не используется*/ false);

// Используем итератор для цикла по всем изображениям и выполняем с ними
// некоторое действие
var image;
while((image = images.nextNode()) != null) {
    image.style.visibility = "hidden";  // Здесь выполняется обработка документа
}
```

17.5.2. DOM Range API

DOM Range API состоит из одного интерфейса, Range. Объект Range представляет непрерывную область[1] содержимого документа, находящегося между указанными начальными и конечными позициями. Многие приложения, отображающие текст и документы, дают пользователю возможность выбрать часть документа путем перемещения указателя мыши при нажатой клавише. Выбранная часть документа концептуально эквивалентна области.[2] Когда узел в дереве документа находится внутри области, мы часто говорим, что узел «выбран», даже несмотря на то, что объект Range может не иметь ничего общего с действием конечного пользователя по выбору части документа. Когда начальная и конечная позиции области совпадают, мы говорим, что эта область «свернута». В этом случае объект Range представляет собой одну позицию или точку вставки внутри документа.

Объект Range предоставляет методы для определения начальной и конечной позиций области, копирования и удаления ее содержимого, вставки узлов в начальную позицию области. Поддержка Range API может отсутствовать. На момент написания этой книги он поддерживался в Netscape 6.1. IE 5 поддерживает собственный API, похожий на Range API, но не совместимый с ним. Поддержка Range API позволяет проверить, например, следующий код:

---

[1] То есть логически непрерывная область. В двунаправленных языках, таких как арабский или иврит, логически непрерывная область может не быть визуально непрерывной при отображении.

[2] Несмотря на то что веб-броузеры обычно разрешают пользователю выделять часть содержимого документа, текущий стандарт DOM Level 2 не делает содержимое этих частей доступным в JavaScript, поэтому нет стандартного способа получить объект Range, соответствующий выделенной пользователем области.

```
document.implementation.hasFeature("Range", "2.0");
                                // True, если Range API поддерживается
```

17.5.2.1. Начальная и конечная позиции

Начальная и конечная позиции области задаются двумя значениями. Первое значение – это узел документа, обычно объект Document, Element или Text. Второе значение – это число, обозначающее позицию внутри этого узла. Если узлом является документ или элемент, то число представляет собой позицию между дочерними узлами документа или элемента. Так, смещение, равное 0, обозначает позицию непосредственно перед первым дочерним узлом данного узла. Смещение, равное 1, обозначает позицию после первого дочернего узла и перед вторым. Если указанный узел является узлом Text (или другим текстовым типом узла, например Comment), то число обозначает позицию между символами текста. Смещение, равное 0, задает позицию перед первым символом текста, равное 1 – позицию между первым и вторым символом и т. д. Если начальная и конечная позиции указаны подобным образом, то область представляет собой все узлы и/или символы между начальной и конечной позициями. Реальная сила интерфейса Range состоит в том, что начальная и конечная позиции могут находиться внутри разных узлов документа, и, следовательно, область может охватывать несколько узлов Element и Text (и их частей).

Для того чтобы продемонстрировать действие различных методов работы с областями, я собираюсь воспользоваться нотацией, применяемой в спецификации DOM для иллюстрации содержимого документа, представляемого областью. Содержимое документа показано в виде исходного кода HTML, в котором содержимое области выделено **жирным шрифтом**. Например, следующая строка показывает область, начинающуюся с нулевой позиции в узле `<body>` и продолжающуюся до позиции 8 в узле Text, содержащемся внутри узла `<h1>`:

```
<body><h1>Document Title</h1><body>
```

Для создания объекта Range вызовите метод `createRange()` объекта Document:

```
var r = document.createRange();
```

В только что созданных областях начало и конец указывают на нулевую позицию объекта Document. Прежде чем вы сможете делать что-нибудь интересное с областью, вы должны установить начальную и конечную позиции для определения нужной области документа. Для этого имеется несколько способов. Самый общий состоит в вызове методов `setStart()` и `setEnd()` для задания начальной и конечной точек. Каждому из методов передаются узел и позиция внутри узла.

Более высокоуровневая технология установки начальной и/или конечной позиции состоит в вызове метода `setStartBefore()`, `setStartAfter()`, `setEndBefore()` или `setEndAfter()`. Каждый из этих методов принимает в качестве аргумента один узел. Они устанавливают начальную или конечную позицию области равной позиции до или после указанного узла внутри родительского.

И наконец, для того чтобы определить область, представляющую один узел или поддерево документа, можно применить методы selectNode() и selectNodeContent(). Оба метода принимают один аргумент-узел. Метод selectNode() устанавливает начальную и конечную позиции до и после указанного узла внутри его родителя, определяя область, включающую узел и все его дочерние узлы. selectNodeContent() устанавливает начало области в позиции перед первым дочерним узлом данного узла, а конец – в позиции после последнего дочернего узла данного узла. Полученная область содержит все дочерние узлы данного узла, но не сам узел.

17.5.2.2. Работа с областями

Определив область, можно сделать с ней несколько интересных вещей. Чтобы удалить содержимое документа внутри области, просто вызовите метод deleteContents() объекта Range. Если область включает частично выбранные узлы Text, то операция удаления оказывается несколько более сложной. Рассмотрим следующую область:

```
<p>This is <i>only</i> a test
```

После вызова метода deleteContents() выделенная часть документа будет выглядеть так:

```
<p>This<i>ly</i> a test
```

Даже несмотря на то что элемент <i> (частично) содержался в области, после операции удаления элемент сохраняется (с измененным содержимым) в дереве документа.

Если необходимо удалить содержимое области из документа, но при этом сохранить удаленное содержимое (возможно, для повторной вставки), то вместо deleteContents() следует применить метод extractContent(). Этот метод удаляет узлы из дерева документа и вставляет их в возвращаемый методом объект DocumentFragment (рассмотренный ранее в этой главе). Если область включает часть узла, этот узел остается в дереве документа, а его содержимое изменяется. Для вставки в объект DocumentFragment создается (и изменяется) копия узла (см. Node.cloneNode()). Снова рассмотрим предыдущий пример. Если вместо deleteContents() вызывается extractContents(), то с документом происходит то же самое, что было показано выше, а возвращаемый объект DocumentFragment содержит:

```
is <i>on</i>
```

Метод extractContents() работает, когда требуется выполнить с документом эквивалент операции «Вырезать». Если же надо выполнить операцию «Скопировать» и извлечь содержимое без удаления его из документа, используйте вместо extractContents() метод cloneContents().[1]

---

[1] Реализация операций «вырезать», «скопировать» и «вставить» в стиле текстовых процессоров реально более сложна. Простые операции с областями на сложном дереве документа не всегда реализуют нужное поведение «вырезать и вставить» на линейном представлении документа.

Начальная позиция области может использоваться не только для указания границ удаляемого или клонируемого текста, но и для задания места вставки внутри документа. Метод `insertNode()` области вставляет указанный узел (и все его дочерние элементы) в документ в начальной позиции области. Если указанный узел уже является частью дерева документа, он перемещается со своего текущего местоположения в позицию, заданную областью. Если указанный узел является объектом DocumentFragmemt, то вместо самого узла вставляются все его дочерние узлы.

Другой полезный метод объекта Range – `surroundContents()`. Этот метод меняет родителя для области на указанный в аргументе узел и вставляет этот узел в дерево документа вместо области. Например, передав функции `surroundContents()` только что созданный узел `<i>`, можно преобразовать эту область

```
This is only a test
```

в

```
This is <i>only</i> a test
```

Обратите внимание, что поскольку открывающие и закрывающие теги в файле HTML должны быть корректно вложенными, `surroundContents()` не может применяться (и генерирует исключение) для областей, частично содержащих какие-либо узлы, отличающиеся от узлов Text. Так, если область ранее служила для иллюстрации метода `deleteContents()`, то она не может использоваться с `surroundContents()`.

В объекте Range также имеются и другие возможности. Можно сравнить границы двух различных областей с помощью `compareBoundaryPoints()`, копировать область с помощью `cloneRange()` и извлекать текстовую копию содержимого области (без разметки) с помощью `toString()`. Начальная и конечная позиции области могут быть прочитаны из доступных только для чтения свойств `startContainer`, `startOffset`, `endContainer` и `endOffset`. Начальная и конечная точки всех допустимых областей имеют общего предка где-то в дереве документа, даже если это объект Document, находящийся в корне дерева. Этого общего предка можно определить с помощью свойства `commonAncestorContainer` объекта Range.

18

Каскадные таблицы стилей и Dynamic HTML

Каскадные таблицы стилей (Cascading Style Sheets, CSS) – это стандарт для задания визуального представления[1] документов HTML (или XML). Теоретически при помощи разметки HTML следует задавать структуру документа, сопротивляясь искушению применять устаревшие теги HTML, такие как `<font>`. Потому что для задания стилей существуют таблицы CSS, определяющие, как именно будут отображаться структурные элементы документа. Например, CSS позволяет указать, что заголовки первого уровня, определяемые тегами `<h1>`, должны отображаться в верхнем регистре, жирными буквами размером в 24 пункта, шрифтом sans-serif, с выравниванием по центру.

Технология CSS ориентирована на дизайнеров и всех, кто заботится о точном визуальном отображении HTML-документов. Она интересна программистам, использующим клиентский JavaScript, так как объектная модель документа позволяет при помощи сценариев применять стили к индивидуальным элементам документа. Совместное применение CSS и JavaScript обеспечивает создание множества визуальных эффектов, неточно называемых Dynamic HTML (DHTML).[2]

Возможность работы со стилями CSS в сценариях позволяет динамически изменять цвета, шрифты и прочие элементы оформления. Еще важнее, что стили CSS позволяют устанавливать и изменять размер элементов и даже скрывать и показывать их. Это значит, что приемы DHTML могут применяться для создания анимированных переходов, когда содержимое доку-

---

[1] А в стандарте CSS2 еще и звукового представления.

[2] Многие сложные эффекты DHTML также используют приемы обработки событий, которые мы рассмотрим в главе 19.

мента «выплывает» справа, или, например, разворачивающегося и сворачивающегося структурированного списка, в котором пользователь может управлять объемом показываемой ему информации.

Эта глава начинается с обзора таблиц стилей CSS и применения стилей CSS для указания позиции и видимости элементов документа. Затем объясняется, как в сценариях работать со стилями CSS с помощью API, определенного стандартом DOM Level 2. И наконец, демонстрируются нестандартные, зависящие от броузера API, позволяющие создавать эффекты DHTML в Netscape 4 и Internet Explorer 4.

18.1. Стили и таблицы стилей в CSS

Стили в CSS задаются как разделенные точкой с запятой пары атрибутов, в которых имя и значение разделены двоеточием. Например, следующий стиль определяет полужирный, подчеркнутый текст синего цвета:

```
font-weight: bold; color: blue; text-decoration: underline;
```

Стандарт CSS определяет много атрибутов стилей, которые можно устанавливать. В табл. 18.1 перечислены все атрибуты, кроме тех, которые втречаются только в звуковых таблицах стилей. Возможно, вы не понимаете или не знакомы со всеми этими атрибутами, их значениями или смыслом. Однако когда вы познакомитесь с CSS и станете применять их в документах и сценариях, эта таблица может оказаться полезной в качестве справочника. Более полную документацию по CSS можно найти в книгах издательства O'Reilly «Cascading Style Sheets: The Definitive Guide» Эрика Мейера (Eric Meyer) или «Dynamic HTML: The Definitive Guide» Денни Гудмена (Danny Goodman). Либо прочтите спецификацию, расположенную по адресу *http://www.w3c.org/TR/REC-CSS2/*.

Во втором столбце табл. 18.1 показаны допустимые значения для каждого атрибута стиля. Здесь используется та же грамматика, что и в спецификации CSS. Значения, указанные моноширинным шрифтом, – это ключевые слова, и они должны присутствовать в документе в том же виде, в каком они приведены в таблице. Значения, выделенные *курсивом*, указывают тип данных, например «строка» или «длина». Обратите внимание, что тип *length* – это число, за которым следует спецификация единицы измерения, например px (пикселы). Описания других типов можно найти в литературе по CSS. Значения, указанные *моноширинным курсивом*, определяют набор значений, допустимых для некоторого другого атрибута CSS. Кроме значений, показанных в таблице, каждый атрибут стиля может иметь значение inherit (наследует), указывающее, что атрибут должен наследовать значение от родительского элемента.

Значения, разделенные символом |, являются альтернативными – надо указать только одно из них. Значения, разделенные символом ||, представляют собой варианты – необходимо указать хотя бы одно из них, но можно указать несколько (в любом порядке). Квадратные скобки [ ] предназначены для объединения значений в группы. Звездочка (*) обозначает, что предыдущее

значение или группа может присутствовать ноль или более раз, знак (+) – что предыдущее значение или группа может присутствовать один или более раз, а вопросительный знак (?) указывает, что предыдущее значение не обязательно и может присутствовать ноль или более раз. Число в фигурных скобках задает количество повторений. Например, {2} задает, что предыдущее значение должно быть повторено дважды, а {1,4} – что предыдущее значение должно присутствовать не менее одного раза и не более четырех раз. (Этот синтаксис повторения может показаться вам знакомым: он совпадает с используемым в регулярных выражениях JavaScript, описанных в главе 10.)

Таблица 18.1. Атрибуты стилей CSS и их значения

Имя	Значение
background	[*background-color* ‖ *background-image* ‖ *background-repeat* ‖ *background-attachment* ‖ *background-position*]
background-attachment	scroll | fixed
background-color	***color*** | transparent
background-image	url(***url***) | none
background-position	[[***percentage*** | ***length***]{1,2} | [[top | center | bottom] ‖ [left | center | right]]]
background-repeat	repeat | repeat-x | repeat-y | no-repeat
border	[*border-width* ‖ *border-style* ‖ ***color***]
border-collapse	collapse | separate
border-color	***color***{1,4} | transparent
border-spacing	***length length***?
border-style	[none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset]{1,4}
border-top border-right border-bottom border-left	[*border-top-width* ‖ *border-top-style* ‖ ***color***]
border-top-color border-right-color border-bottom-color border-left-color	***color***
border-top-style border-right-style border-bottom-style border-left-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
border-top-width border-right-width border-bottom-width border-left-width	thin | medium | thick | ***length***
border-width	[thin | medium | thick | ***length***]{1,4}
bottom	***length*** | ***percentage*** | auto

Имя	Значение
caption-side	top | bottom | left | right
clear	none | left | right | both
clip	[rect([*length* | auto]{4})] | auto
color	*color*
content	[*string* | url(*url*) | *counter* | attr(*attribute-name*) | open-quote | close-quote | no-open-quote | no-close-quote]+
counter-increment	[*identifier integer*?]+ | none
counter-reset	[*identifier integer*?]+ | none
cursor	[[url(*url*) ,]* [auto | crosshair | default | pointer | move | e-resize | ne-resize | nw-resize | n-resize | se-resize | sw-resize | s-resize | w-resize | text | wait | help]]
direction	ltr | rtl
display	inline | block | list-item | run-in | compact | marker | table | inline-table | table-row-group | table-header-group | table-footer-group | table-row | table-column-group | table-column | table-cell | table-caption | none
empty-cells	show | hide
float	left | right | none
font	[[*font-style* ‖ *font-variant* ‖ *font-weight*]? *font-size* [/ *line-height*]? *font-family*] | caption | icon | menu | message-box | small-caption | status-bar
font-family	[[*family-name* | serif | sans-serif | monospace | cursive | fantasy],]+
font-size	xx-small | x-small | small | medium | large | x-large | xx-large | smaller | larger | *length* | *percentage*
font-size-adjust	*number* | none
font-stretch	normal | wider | narrower | ultra-condensed | extra-condensed | condensed | semi-condensed | semi-expanded | expanded | extra-expanded | ultra-expanded
font-style	normal | italic | oblique
font-variant	normal | small-caps
font-weight	normal | bold | bolder | lighter | 100 | 200 | 300 | 400 | 500 | 600 | 700 | 800 | 900
height	*length* | *percentage* | auto
left	*length* | *percentage* | auto
letter-spacing	normal | *length*
line-height	normal | *number* | *length* | *percentage*
list-style	[*list-style-type* ‖ *list-style-position* ‖ *list-style-image*]
list-style-image	url(*url*) | none
list-style-position	inside | outside
list-style-type	disc | circle | square | decimal | decimal-leading-zero | lower-roman | upper-roman | lower-greek | lower-alpha | lower-latin | upper-alpha | upper-latin | hebrew | armenian | georgian | cjk-ideographic | hiragana | katakana | hiragana-iroha | katakana-iroha | none

Таблица 18.1 (продолжение)

Имя	Значение
margin	[*length* | *percentage* | auto]{1,4}
margin-top margin-right margin-bottom margin-left	*length* | *percentage* | auto
marker-offset	*length* | auto
marks	[crop ‖ cross] | none
max-height	*length* | *percentage* | none
max-width	*length* | *percentage* | none
min-height	*length* | *percentage*
min-width	*length* | *percentage*
orphans	*integer*
outline	[*outline-color* ‖ *outline-style* ‖ *outline-width*]
outline-color	*color* | invert
outline-style	none | hidden | dotted | dashed | solid | double | groove | ridge | inset | outset
outline-width	thin | medium | thick | *length*
overflow	visible | hidden | scroll | auto
padding	[*length* | *percentage*]{1,4}
padding-top padding-right padding-bottom padding-left	*length* | *percentage*
page	*identifier* | auto
page-break-after	auto | always | avoid | left | right
page-break-before	auto | always | avoid | left | right
page-break-inside	avoid | auto
position	static | relative | absolute | fixed
quotes	[*string string*]+ | none
right	*length* | *percentage* | auto
size	*length*{1,2} | auto | portrait | landscape
table-layout	auto | fixed
text-align	left | right | center | justify | *string*
text-decoration	none | [underline ‖ overline ‖ line-through ‖ blink]
text-indent	*length* | *percentage*
text-shadow	none | [*color* ‖ *length length length*? ,]* [*color* ‖ *length length length*?]
text-transform	capitalize | uppercase | lowercase | none
top	*length* | *percentage* | auto
unicode-bidi	normal | embed | bidi-override

Имя	Значение
vertical-align	baseline | sub | super | top | text-top | middle | bottom | text-bottom | *percentage* | *length*
visibility	visible | hidden | collapse
white-space	normal | pre | nowrap
widows	*integer*
width	*length* | *percentage* | auto
word-spacing	normal | *length*
z-index	auto | *integer*

Стандарт CSS позволяет объединять определенные атрибуты стилей, которые часто задаются вместе, с помощью специальных атрибутов-сокращений. Например, атрибуты font-family, font-size, font-style и font-weight могут быть одновременно установлены с помощью одного атрибута font:

```
font: bold italic 24pt helvetica;
```

На самом деле некоторые из атрибутов, перечисленных в табл. 18.1, сами представляют собой сокращения. Атрибуты margin и padding – это сокращения для атрибутов, задающих поля, отступы и границы отдельной стороны элемента. Поэтому вместо атрибута margin можно задать атрибуты margin-left, margin-right, margin-top и margin-bottom. То же самое относится и к padding.

18.1.1. Применение правил стиля к элементам документа

Атрибуты стиля применяются к элементам документа несколькими способами. Один из способов состоит в том, чтобы указать их в атрибуте style тега HTML. Например, поля отдельного абзаца можно установить так:

```
<p style="margin-left: 1in; margin-right: 1in;">
```

Одной из главных задач CSS является отделение содержимого и структуры документа от его представления. Задание стилей с помощью атрибута style в отдельных тегах HTML не способствует решению этой задачи (хотя может быть полезным приемом для DHTML). Чтобы добиться разделения структуры и представления, применяются *таблицы стилей*, собирающие всю информацию о стилях в одном месте. Таблица стилей CSS состоит из набора правил стиля. Каждое правило начинается с селектора, задающего элемент или элементы документа, к которым применяется правило; за селектором следует набор атрибутов стиля и их значений, заключенный в фигурные скобки. Простейший вид правил определяет стиль для одного или нескольких определенных тегов. Например, следующее правило устанавливает поля и цвет фона для тега <body>:

```
body { margin-left: 30px; margin-right: 15px; background-color: #ffffff }
```

Следующее правило определяет, что текст внутри заголовков <h1> и <h2> должен быть выровнен по центру:

```
h1, h2 { text-align: center; }
```

Обратите внимание на использование в примере запятой для разделения имен тегов, к которым должны применяться стили. Если запятая отсутствует, селектор задает контекстное правило, применяемое только тогда, когда один тег вложен в другой. Например, следующие правила указывают, что теги <blockquote> отображаются курсивом, но текст внутри тега <i>, находящегося внутри <blockquote>, отображается прямым шрифтом:

```
blockquote { font-style: italic; }
blockquote i { font-style: normal; }
```

Другой тип правил в таблице стилей задает *классы* элементов, к которым должны применяться его стили, и селектор в этом случае иной. Класс элемента определяется атрибутом class тега HTML. Например, следующее правило указывает, что любой тег с атрибутом class="attention" должен отображаться полужирным шрифтом:

```
.attention { font-weight: bold; }
```

Селекторы классов могут объединяться с селекторами имен тегов. Следующее правило указывает, что если в теге <p> есть атрибут class="attention", тег должен отображаться красным цветом (кроме того что полужирным шрифтом, как определено предыдущим правилом):

```
p.attention { color: red; }
```

И наконец, таблицы стилей содержат правила, применяемые к отдельным элементам, имеющим заданное значение атрибута id. Следующее правило указывает, что элемент с атрибутом id, равным «p1», не должен быть показан:

```
#p1 { visibility: hidden; }
```

Мы встречали атрибут id раньше: он применяется с функцией DOM getElementById() для получения отдельных элементов документа. Как можно предположить, этот вид правил в таблице стилей полезен, если надо управлять стилем отдельного элемента. При наличии такого правила сценарий может менять значение атрибута visibility с hidden на visible, вызывая динамическое появление элемента.

18.1.2. Связывание таблиц стилей с документами

Таблицу стилей можно внедрить в HTML-документ, поместив ее между тегами <style> и </style> в заголовке документа, или сохранить ее в собственном файле и сослаться на файл из HTML-документа с помощью тега <link>. Эти две технологии можно объединить, создавая специфическую для документа таблицу стилей между тегами <style>, которая ссылается на независимую

таблицу стилей или импортирует ее с помощью специального «@-правила» `@import`. Детальное описание `@import` можно найти в литературе по CSS.

18.1.3. Каскад

Вспомните, что буква «C» в CSS обозначает «cascade» (каскадная). Этот термин указывает, что правила стилей, применяемые к конкретному элементу документа, могут быть получены из каскада различных источников. Каждый веб-броузер имеет свои собственные стили по умолчанию для всех элементов HTML и может разрешать пользователю перекрывать эти значения с помощью пользовательской таблицы стилей. Автор документа может определять таблицы стилей с помощью тегов `<style>` или внешних файлов, связанных с другими таблицами стилей или импортированных в них. Автор также может определять встроенные стили для индивидуальных элементов с помощью атрибута `style`.

Спецификация CSS включает полный набор правил, определяющих, какие правила из каскада имеют приоритет над другими. Однако если вкратце, то следует знать, что пользовательская таблица стилей перекрывает таблицу стилей броузера по умолчанию, авторская таблица стилей перекрывает пользовательскую таблицу стилей, а встроенные стили перекрывают все. Исключение из этого общего правила состоит в том, что пользовательские атрибуты стилей, значения которых включают модификатор `important`, перекрывают авторские стили. Если в таблице стилей к элементу применяется более одного правила, то стили, определенные по наиболее конкретному правилу, перекрывают конфликтующие стили, определенные по менее конкретным правилам. Правила, указывающие `id` элемента, являются наиболее конкретными. Правила, указывающие `class`, – следующие по конкретности. Правила, указывающие только имена тегов, – наименее конкретные, но правила, указывающие несколько имен вложенных тегов, более конкретны, чем правила, указывающие только одно имя тега.

18.1.4. Версии CSS

На момент написания этой книги существовало две версии стандарта CSS. CSS1 был принят в декабре 1996 года и определяют атрибуты для задания цветов, шрифтов, полей, границ и других базовых стилей. Netscape 4 и Internet Explorer 4, по крайней мере частично, реализуют CSS1. Второе издание стандарта, CSS2, было принято в мае 1998 года; оно определяет более развитые возможности, наиболее важная из которых – абсолютное позиционирование элементов. Развитые возможности CSS2 поддерживаются только в броузерах шестого поколения. Однако, к счастью, процесс стандартизации ключевых возможностей позиционирования из CSS2 был начат как часть отдельного проекта CSS-Positioning (CSS-P), и, следовательно, некоторые из средств DHTML доступны в броузерах четвертого поколения. Продолжается работа над третьим изданием стандарта CSS. Спецификации и рабочие проекты CSS можно найти по адресу *http://www.w3.org/Style/CSS/*.

18.1.5. Пример CSS

Пример 18.1 представляет собой HTML-файл, определяющий и использующий таблицу стилей. Он демонстрирует описанные выше правила стилей, базирующиеся на имени тега, классе и идентификаторе, а также содержит пример встроенного стиля, определяемого атрибутом `style`. Помните, что этот пример приведен здесь только в качестве обзора синтаксиса и возможностей CSS. Полное описание CSS выходит за рамки этой книги.

Пример 18.1. Определение и использование каскадных таблиц стилей

```
<head>
<style type="text/css">
/* Указывает, что заголовки отображаются курсивом синего цвета. */
h1, h2 { color: blue; font-style: italic }

/*
 * Любой элемент класса WARNING отображается крупными жирными символами,
 * имеет большие поля и желтый фон с жирной красной границей.
 */
.WARNING {
          font-weight: bold;
          font-size: 150%;
          margin: 0 1in 0 1in; /* сверху справа, снизу слева */
          background-color: yellow;
          border: solid red 8px;
          padding: 10px;       /* 10 пикселов со всех 4 сторон */
}

/*
 * Текст заголовков h1 и h2 внутри элементов с class="WARNING"
 * должен быть выровнен по центру, в дополнение к выделению синим курсивом.
 */
.WARNING h1, .WARNING h2 { text-align: center }

/* Отдельный элемент с id="P23" отображается прописными буквами по центру. */
#P23 {
      text-align: center;
      text-transform: uppercase;
}
</style>
</head>
<body>
<h1>Демонстрация использования каскадных таблиц стилей </h1>

<div class="WARNING">
<h2>Предупреждение</h2>
Это предупреждение!
Заметьте, как оно привлекает ваше внимание жирным шрифтом и яркими цветами.
Также обратите внимание, что заголовок выровнен по центру и выделяется
синим курсивом.
```

```
</div>

<p id="P23">
Этот абзац выровнен по центру<br>
и выводится прописными буквами.<br>
<span style="text-transform: none">
Здесь мы явно используем встроенный стиль для перекрытия прописных букв.
</span>
</p>
</body>
```

18.2. Позиционирование элемента с помощью CSS

Для разработчиков содержимого DHTML в CSS важнее всего то, что таблицы стилей позволяют указывать видимость, размер и точную позицию отдельных элементов документа посредством обычных атрибутов стилей CSS. Занимаясь программированием DHTML, важно понимать, как работают эти атрибуты стилей. Они перечислены в табл. 18.2, а в последующих разделах описаны более подробно.

Таблица 18.2. Атрибуты позиционирования и видимости в CSS

Атрибуты	Описание
`position`	Задает тип позиционирования, применяемый к элементу
`top`, `left`	Задает позицию верхнего и левого краев элемента
`bottom`, `right`	Задает позицию нижнего и правого краев элемента
`width`, `height`	Задает размер элемента
`z-index`	Задает для элемента «порядок в стеке» относительно любых перерывающих его элементов; определяет третье измерение в позиционировании элемента
`display`	Определяет, отображается ли элемент и как он отображается
`visibility`	Определяет, является ли элемент видимым
`clip`	Определяет «область отсечения» элемента; отображаются только те части документа, которые находятся внутри этой области
`overflow`	Определяет, что следует делать, если размер элемента больше, чем предоставленное ему место

18.2.1. Ключ к DHTML: атрибут position

Атрибут CSS `position` задает тип позиционирования, применяемый к элементу. Вот четыре возможных значения этого атрибута:

`static`

Это значение по умолчанию и оно указывает, что элемент позиционируется в соответствии с нормальным потоком содержимого документа (для большинства западных языков – слева направо и сверху вниз). Статичес-

ки позиционированные элементы не являются элементами DHTML и не могут позиционироваться с помощью top, left и других атрибутов. Для использования технологии позиционирования DHTML с элементом документа сначала необходимо установить его атрибут position равным одному из трех других значений.

absolute

Это значение позволяет задавать абсолютную позицию элемента относительно содержащего его элемента. Такие элементы позиционируются независимо от всех остальных элементов и не являются частью потока статически позиционированных элементов. Абсолютно позиционированный элемент позиционируется либо относительно тела документа, либо, если он вложен в другой абсолютно позиционированный элемент, относительно этого элемента. Это наиболее часто применяемый в DHTML тип позиционирования.

fixed

Это значение позволяет указать положение элемента относительно окна броузера. Элементы с фиксированным позиционированием не прокручиваются с остальной частью документа и поэтому могут использоваться для имитации фреймов. Как и абсолютно позиционированные, фиксировано позиционированные элементы не зависят от всех остальных элементов и не являются частью потока документа. Фиксированное позиционирование относится к средствам CSS2 и не поддерживается броузерами четвертого поколения. (Оно поддерживается в Netscape 6 и IE 5 для Macintosh, но не поддерживается в IE 5 и IE 6 для Windows).

relative

Если атрибут position установлен в relative, элемент располагается в соответствии с нормальным потоком, а затем его положение смещается относительно его обычного положения в потоке. Пространство, выделенное для элемента в нормальном потоке документа, остается выделенным для него, и элементы, расположенные со всех сторон от него, не смыкаются для заполнения этого пространства и не выталкиваются с новой позиции элемента. Относительное позиционирование может быть полезно для некоторых статических дизайнерских целей, но редко применяется для создания эффектов DHTML.

18.2.2. Задание положения и размера элементов

Установив для атрибута position элемента значение, отличное от static, можно задать положение элемента с помощью произвольной комбинации атрибутов left, top, right и bottom. Наиболее распространенный прием позиционирования – это указание атрибутов left и top, задающих расстояние от левого края элемента-контейнера (обычно самого документа) до левого края позиционируемого элемента и расстояние от верхнего края контейнера до верхнего края элемента. Так, чтобы поместить элемент на расстоя-

нии 100 пикселов от левого края и 100 пикселов от верхнего края документа, можно задать стили CSS в атрибуте style следующим образом:

```
<div style="position: absolute; left: 100px; top: 100px;">
```

Элемент-контейнер, относительно которого позиционируется динамический элемент, не обязательно совпадает с элементом-контейнером, внутри которого элемент определен в исходном тексте документа. Динамические элементы не являются частью обычного потока элементов, поэтому их положение не задается относительно статических элементов-контейнеров, внутри которых они определены. Большинство динамических элементов позиционируются относительно самого документа (тега <body>). Исключение составляют динамические элементы, определенные внутри других динамических элементов. В этом случае вложенный динамический элемент позиционируется относительно ближайшего динамического предка.

Чаще всего задается положение верхнего левого угла элемента с помощью атрибутов left и top, но можно задавать положение нижнего и правого краев элемента относительно нижнего и правого краев элемента-контейнера в атрибутах right и bottom. Например, при помощи такого стиля можно указать, чтобы правый нижний угол элемента находился в правом нижнем углу документа (предполагая, что он не вложен в другой динамический элемент):

```
position: absolute; right: 0px; bottom: 0px;
```

Для того чтобы верхний край элемента располагался в 10 пикселах от верхнего края окна, а правый в 10 пикселах от правого края окна, можно применить следующие стили:

```
position: absolute; right: 10px; bottom: 10px;
```

Обратите внимание, что атрибуты right и bottom — это новые добавления к стандарту CSS, которые не поддерживаются броузерами четвертого поколения, как атрибуты top и left.

Кроме позиции элементов, CSS позволяет указывать их размер. Чаще всего это делается путем задания значений атрибутов стиля width и height. Например, следующий код HTML создает абсолютно позиционированный элемент без содержимого. Значения атрибутов width, height и background-color заданы так, чтобы он отображался в виде маленького синего квадрата:

```
<div style="position: absolute; left: 10px; right: 10px;
            width: 10px; height: 10px; background-color: blue">
</div>
```

Другой способ определения ширины элемента состоит в одновременном задании атрибутов left и right. Аналогично можно задать высоту элемента, одновременно указав оба атрибута, top и bottom. Однако если задать значения для left, right и width, то атрибут width перекрывает атрибут right, а если ограничивается высота элемента, то атрибут height имеет приоритет перед bottom.

Имейте в виду, что необязательно задавать размер для каждого динамического элемента. Некоторые элементы, такие как изображения, имеют присущий им размер. Кроме того, для динамических элементов, включающих текст или другое потоковое содержимое, часто достаточно указать желаемую ширину элемента и разрешить автоматическое определение высоты в зависимости от размещения содержимого элемента.

В предыдущих примерах значения атрибутов позиционирования и размера задавались с суффиксом «px». Он означает «pixels» (пикселы). Стандарт CSS допускает указание размерности в некоторых других единицах, в том числе в дюймах («in»), сантиметрах («cm»), пунктах («pt») и в em («em» – это единица измерения высоты строки текущего шрифта).[1] Пикселы – это наиболее часто используемые в программировании DHTML единицы измерения. Обратите внимание, что стандарт CSS требует указания единицы измерения. Некоторые броузеры могут предполагать пикселы, если единица измерения не указана, но на это не следует особенно полагаться.

CSS также позволяет задавать положение и размер элемента в процентах от размера элемента-контейнера и в абсолютных единицах с помощью описанных выше единиц измерения. Следующий код HTML создает пустой элемент с черной границей, имеющий ширину и высоту в половину элемента-контейнера (или окна броузера) и расположенный по центру в этом элементе:

```
<div style="position: absolute; left: 25%; top: 25%; width: 50%; height: 50%;
            border: 2px solid black">
</div>
```

18.2.2.1. Еще о размере и положении элемента

Важно понимать некоторые детали работы атрибутов `left`, `right`, `width`, `top`, `bottom` и `height`. Во-первых, `width` и `height` задают только размер области содержимого элемента и не включают какое-либо дополнительное пространство, требуемое для отступов, рамки или полей элемента. Для того чтобы определить полный экранный размер элемента с границей, следует прибавить к ширине элемента величину отступов слева и справа, а также ширину левой и правой рамки, а к высоте элемента добавить величину отступов сверху и снизу, а также ширину нижней и верхней рамки.

Атрибуты `width` и `height` определяют только область содержимого элемента, поэтому можно подумать, что `left` и `height` (а также `right` и `bottom`) будут измеряться относительно области содержимого элемента-контейнера. На самом деле стандарт CSS указывает, что эти значения измеряются относительно внешнего края отступов элемента-контейнера (совпадающего с внутренним краем рамки элемента).

---

[1] Пункт в системе Дидо (Европа, Россия) равен 0,376 мм, а в системе Пика (США) – 0,351 мм. em – круглая шпация, примерно равная кеглю шрифта (или ширине литеры m). – *Примеч. ред.*

Чтобы сказанное было понятнее, рассмотрим пример. Предположим, что у вас есть динамически позиционированный элемент, вокруг содержимого которого имеются отступы размером 10 пикселов, а вокруг них – рамка толщиной 5 пикселов. Теперь предположим, что вы динамически позиционируете дочерний элемент внутри этого контейнера. Если установить атрибут `left` дочернего элемента равным «0 px», то обнаружится, что левый край дочернего элемента будет находиться непосредственно у внутреннего края рамки контейнера. При этом значении атрибута дочерний элемент перекрывает отступы контейнера, хотя предполагается, что они остаются пустыми (так как это и есть назначение отступов). Для того чтобы поместить дочерний элемент в левый верхний угол области содержимого контейнера, необходимо установить атрибуты `left` и `top` равными «10 px» (рис. 18.1).

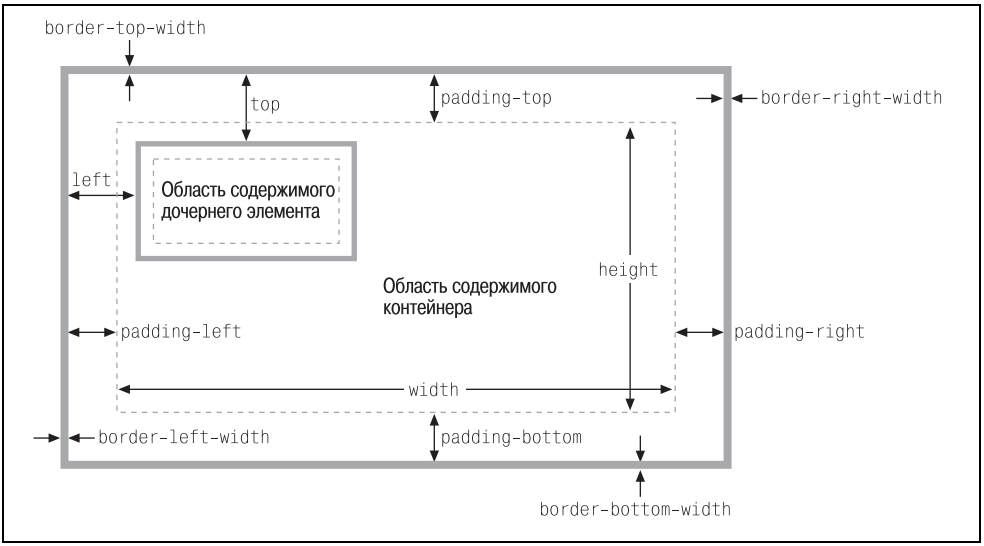

Рис. 18.1. *Динамически позиционированный контейнер и дочерние элементы с некоторыми CSS-атрибутами*

Теперь вы понимаете, что `width` и `height` задают размер только области содержимого элемента и что атрибуты `left`, `top`, `right` и `bottom` измеряются относительно отступов элемента-контейнера, и вам пора узнать еще одну подробность: Internet Explorer версий от 4 до 5.5 для Windows (но не IE 5 для Mac) реализуют атрибуты `width` и `height` некорректно и включают в их значения рамку и отступы (но не поля). Так, если установить ширину элемента равной 100 пикселам и поместить слева и справа поле шириной 10 пикселов и рамку толщиной 5 пикселов, то ширина области содержимого элемента в этих ошибочных версиях Internet Explorer будет равна лишь 70 пикселам.

В IE 6 CSS-атрибуты положения и размера работают корректно, когда броузер находится в стандартном режиме, и некорректно (но совместимо с более ранними версиями), когда броузер находится в режиме совместимости. Стандартный режим и, соответственно, корректная реализация модели CSS

включаются при наличии тега <!DOCTYPE> в начале документа, который объявляет, что документ соответствует стандарту HTML 4.0 (или более поздней версии) или некоторой версии стандартов XHTML. Например, любое из следующих объявлений типа HTML-документа приводит к отображению документов в IE 6 в стандартном режиме:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Strict//EN">
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"
    "http://www.w3.org/TR/html4/loose.dtd">
```

Броузеры Netscape 6 и Mozilla корректно обрабатывают атрибуты width и height. Но в этих броузерах тоже есть стандартный режим и режим совместимости, как и в IE. Отсутствие объявления <!DOCTYPE> переводит броузер в «специальный» режим, в котором он имитирует некоторые (не очень значительные) нестандартные способы размещения из Netscape 4. Наличие <!DOCTYPE> заставляет броузер выйти из режима совместимости с Netscape 4 и корректно реализовывать стандарты.

18.2.3. Третье измерение: z-index

Мы видели, что с помощью атрибутов left, top, right и bottom можно задавать координаты X и Y элементов внутри двумерной плоскости элемента-контейнера. Атрибут z-index определяет что-то вроде третьего измерения – он позволяет задать порядок элементов в стеке и указывает, какой из перекрывающихся элементов будет расположен поверх других. Атрибут z-index представляет собой целое число. По умолчанию он равен нулю, но можно задавать положительные и отрицательные значения (хотя броузеры четвертого поколения могут не поддерживать отрицательные значения z-index). Когда два или более элементов перекрываются, они отрисовываются в порядке от наименьшего к наибольшему z-index; элемент с наибольшим z-index оказывается расположенным поверх всех остальных. Если перекрывающиеся элементы имеют одинаковый z-index, они отрисовываются в том порядке, в котором присутствуют в документе, поэтому наверху оказывается последний из перекрывающихся элементов.

Обратите внимание, что стек z-index применяется только к смежным элементам (то есть к дочерним элементами одного контейнера). Если перекрываются два элемента, не являющиеся смежными, то с полмощью индивидуальных атрибутов z-index нельзя указать, какой из них находится наверху. Вместо этого надо задать атрибут z-index для двух смежных контейнеров двух перекрывающихся элементов.

Непозиционированные элементы всегда размещаются способом, не допускающим перекрывание, поэтому к ним атрибут z-index не применяется. Тем не менее для них значение z-index по умолчанию равно нулю, то есть позиционированные элементы с положительным z-index оказываются расположенными поверх обычного потока документа, а позиционированные элементы с отрицательным z-index оказываются под обычным потоком документа.

И наконец, заметим, что некоторые броузеры не учитывают атрибут z-index, когда он применяется к тегам <iframe>, и вы можете обнаружить, что встраиваемые фреймы располагаются поверх других элементов, независимо от указанного порядка в стеке. Такие же неприятности могут быть и с другими «оконными» элементами, например с выпадающими меню <select>. Броузеры четвертого поколения могут отображать все управляющие элементы форм поверх абсолютно позиционированных элементов, независимо от значений z-index.

18.2.4. Отображение и видимость элемента

Есть два CSS-атрибута, позволяющих управлять видимостью элемента документа: visibility и display. Атрибут visibility прост, – если его значение равно hidden, то элемент не отображается, а если visible, то элемент отображается. Атрибут display более универсален и служит для указания типа отображения элемента. Он определяет, блочный ли это элемент, встраиваемый, элемент списка и т. д. Однако если атрибут display установлен в none, то элемент вообще не отображается и даже не размещается.

Различие между атрибутами стиля visibility и display имеет отношение к их действию на элементы, не позиционированные динамически. Для элемента, расположенного в нормальном потоке документа (с атрибутом position, равным static или relative), установка visibility в none делает элемент невидимым, но резервирует для него место в документе. Такой элемент может повторно скрываться и отображаться без изменения компоновки документа. Однако если атрибут display элемента установлен в none, то место в документе для него не выделяется; элементы с обеих сторон от него смыкаются, как будто его нет. (По отношению к абсолютно или фиксированно позиционированными элементами атрибуты visibility и display имеют одинаковое действие, так как эти элементы все равно никогда не являются частью общей компоновки документа.) Обычно атрибут visibility задается при работе с динамически позиционированными элементами. Атрибут display полезен при создании таких эффектов, как разворачивающиеся и сворачивающиеся структурированные списки.

Обратите внимание, что нет особого смысла использовать атрибуты visibility и display для того, чтобы делать элемент невидимым, если вы не собираетесь динамически устанавливать их при помощи JavaScript, чтобы в какой-то момент сделать его снова видимым![1] Далее в этой главе рассказывается, как это сделать.

---

[1] Из этого правила есть исключение: создавая документ, зависящий от CSS, можно предупредить тех, кто работает с броузерами, не поддерживающими CSS, посредством такого кода:

```
<p style=»display:none»>
Этот документ использует CSS, но ваш броузер их не поддерживает!
</p>
```

18.2.5. Частичная видимость: overflow и clip

Атрибут `visibility` позволяет полностью скрыть элемент документа. С помощью атрибутов `overflow` и `clip` можно отобразить только часть элемента. Атрибут `overflow` указывает, что происходит, когда содержимое документа превышает размер, указанный для элемента (например, в атрибутах стиля `width` и `height`). Вот допустимые значения этого атрибута и их смысл:

`visible`

Содержимое может выходить за пределы и по необходимости отрисовываться вне прямоугольника элемента. Это значение по умолчанию.

`hidden`

Содержимое, вышедшее за пределы элемента, обрезается и скрывается, так что никакая часть содержимого никогда не отрисовывается вне области, определяемой атрибутами размера и позиционирования.

`scroll`

Прямоугольник элемента имеет постоянные горизонтальную и вертикальную полосы прокрутки. Если содержимое превышает размеры прямоугольника, полосы прокрутки позволяют увидеть остальное содержимое. Это значение учитывается, только когда документ отображается на экране компьютера; когда документ выводится, например, на бумагу, полосы прокрутки, очевидно, не имеют смысла.

`auto`

Полосы прокрутки отображаются не постоянно, а только когда содержимое превышает размер элемента.

Если свойство `overflow` позволяет указывать, что происходит, когда содержимое элемента больше прямоугольника элемента, а с помощью свойства `clip` можно точно указать, какая часть элемента должна отображаться, независимо от того, выходит ли содержимое за пределы элемента. Этот атрибут особенно полезен для программирования эффектов DHTML, при которых элемент открывается или отображается постепенно.

Значение свойства `clip` задает область отсечения элемента. В CSS2 области отсечения прямоугольные, но синтаксис атрибута `clip` сохраняет возможность того, что следующие версии стандарта будут поддерживать области отсечения, отличные от прямоугольных. Синтаксис атрибута `clip`:

```
rect(top right bottom left)
```

Значения *top*, *right*, *bottom* и *left* задают границы прямоугольника отсечения относительно левого верхнего угла прямоугольника элемента.[1] Чтобы,

[1] По первоначально написанной спецификации CSS2 эти четыре значения задают смещения краев области отсечения от каждого из соответствующих краев элемента. Однако все основные реализации броузеров делают не так и интерпретируют значения *right* и *bottom* как смещения от левого и верхнего краев. Поскольку реализации согласованно разошлись со спецификацией, изменена была спецификация.

например, вывести только часть элемента 100×100 пикселов, можно задать для этого элемента следующий атрибут `style`:

```
style="clip: rect(0px 100px 100px 0px);"
```

Обратите внимание, что четыре значения в скобках представляют собой значения длины и включают спецификацию единицы измерения, например «px» для пикселов. Проценты здесь не допускаются. Значения могут быть отрицательными и тогда указывают, что область отсечения выходит за пределы прямоугольника, определенного для элемента. Для любого из четырех значений ключевое слово `auto` указывает, что этот край области отсечения совпадает с соответствующим краем самого элемента. Например, можно вывести только левые 100 пикселов элемента с помощью следующего атрибута `style`:

```
style="clip: rect(auto 100px auto auto);"
```

Обратите внимание, что между значениями нет запятых, и края области отсечения задаются по часовой стрелке, начиная с верхнего края.

18.2.6. Пример позиционирования с помощью CSS

В примере 18.2 содержится нетривиальный образец применения таблиц стилей CSS и атрибутов позиционирования CSS. При отображении этого HTML-документа в броузере, совместимом с CSS, создается визуальный эффект «субокон» внутри окна броузера. На рис. 18.2 изображен эффект, создаваемый кодом примера 18.2. Несмотря на то что листинг не содержит кода JavaScript, это полезная демонстрация мощных эффектов, которые могут быть достигнуты с помощью CSS в целом и атрибутов позиционирования CSS в частности.

Пример 18.2. Отображение окон с помощью CSS

```
<head>
<style type="text/css">
/**
 * Это таблица стилей CSS, определяющая три стилевых правила, которые мы
 * используем в теле документа для создания визуального эффекта "окна".
 * Правила используют атрибуты позиционирования для установки общего размера
 * окна и положения его компонентов. Изменение размера окна требует
 * аккуратного изменения атрибутов позиционирования во всех трех правилах.
 **/
div.window {  /* Задает размер и границу окна */
    position: absolute;            /* Положение задается в другом месте */
    width: 300px; height: 200px;   /* Размер окна, не включая границы */
    border: outset gray 3px;       /* Обратите внимание на 3D-эффект "outset" */
}

div.titlebar {  /* Задает положение, размер и стиль заголовка "окна" */
```

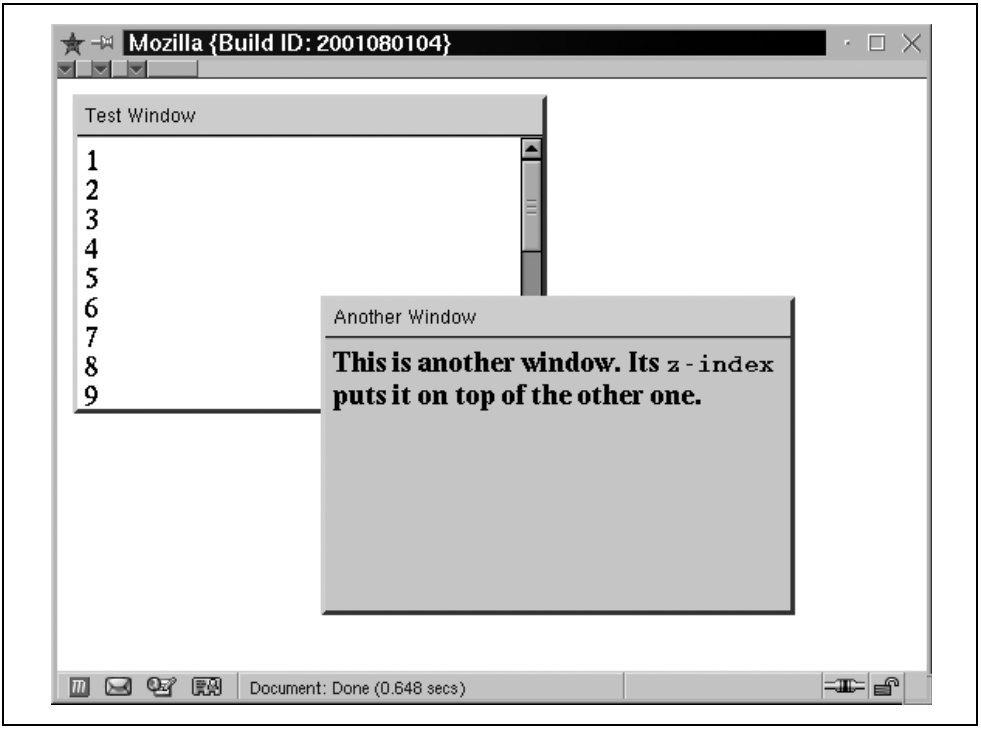

Рис. 18.2. *Окна, созданные с помощью CSS*

```
    position: absolute;         /* Это позиционированный элемент */
    top: 0px; height: 18px;     /* Заголовок имеет размер 18px + */
                                /* + дополняющие пробелы и границы */
    width: 290px;            /* 290 + 5px дополняющие пробелы слева и справа = 300 */
    background-color: ActiveCaption;  /* Используем системный цвет заголовка */
    border-bottom: groove black 2px;  /* Заголовок имеет границу только снизу*/
    padding: 3px 5px 2px 5px;   /* Значения по часовой стрелке: */
                                /* сверху, справа, снизу, слева */
    font: caption;              /* Используем для заголовка системный шрифт */
}

div.content {          /* Задание размера, положения и прокрутки содержимого окна */
    position: absolute;        /* Это позиционированный элемент */
    top: 25px;                 /* 18px заголовок+2px граница+3px+2px доп.пробелы */
    height: 165px;             /* 200px всего - 25px заголовок - 10px доп.пробелы */
    width: 290px;              /* 300px ширина - 10px дополняющие пробелы */
    padding: 5px;              /* Пространство со всех четырех сторон */
    overflow: auto;            /* Если требуются полосы прокрутки, разрешаем их */
    background-color: #ffffff; /* По умолчанию фон белый */
}
</style>
</head>
```

```
<body>
<!-- Вот как мы определяем окно: элемент div "window" с заголовком и -->
<!-- вложенный между ними div для содержимого. Обратите внимание, как -->
<!-- задается позиционирование с помощью атрибута style, дополняющего -->
<!-- стили из таблицы стилей                                          -->
<div class="window" style="left: 10px; top: 10px; z-index: 10;">
<div class="titlebar">Тестовое окно</div>
<div class="content">
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- Много строк для   -->
1<br>2<br>3<br>4<br>5<br>6<br>7<br>8<br>9<br>0<br> <!-- демонстрации прокрутки-->
</div>
</div>

<!-- Это еще одно окно с другой позицией, цветом и шрифтом -->
<div class="window" style="left: 170px; top: 140px; z-index: 20;">
<div class="titlebar">Еще одно окно</div>
<div class="content" style="background-color:#d0d0d0; font-weight:bold;">
Это еще одно окно. Его <tt>z-index</tt> помещает это окно поверх другого.
</div>
</div>
</body>
```

Основной недостаток этого примера в том, что таблица стилей задает фиксированный размер для всех окон. Так как заголовок и содержимое окна должны точно позиционироваться внутри окна в целом, изменение размера окна требует изменения значения различных атрибутов позиционирования во всех трех правилах, определенных в таблице стилей. Это трудно сделать в статическом HTML-документе, но это будет не так трудно, если мы сможем установить все необходимые атрибуты при помощи сценария. Мы изучим эту тему в следующем разделе.

18.3. Использование стилей в сценариях

Главное в DHTML – возможность динамически изменять атрибуты стиля, применяемые к отдельным элементам документа, при помощи JavaScript. Стандарт DOM Level 2 определяет API, позволяющий довольно легко это делать. В главе 17 рассматривалось применение DOM API для получения ссылок на элементы документа либо по имени тега или по ID, либо рекурсивно, обходом всего документа. Получив ссылку на элемент, со стилями которого вы хотите работать, вы устанавливаете свойство `style` элемента, чтобы получить объект CSS2Properties для данного элемента документа. Этот объект JavaScript имеет свойства, соответствующие всем атрибутам стиля CSS1 и CSS2. Установка этих свойств имеет тот же эффект, что и установка соответствующих стилей в атрибуте `style` данного элемента. Чтение этих свойств возвращает значение атрибута CSS, которое, возможно, было установлено в атрибуте `style` элемента. Важно понимать, что полученный вами объект CSS2Properties со свойством `style` элемента определяет только встроенные стили элемента. Невозможно использовать свойства объекта CSS2Properties

для получения информации о примененных к элементу стилях из таблицы стилей. Устанавливая свойства для этого объекта, вы определяете встроенные стили, перекрывающие стили из таблицы стилей.

Рассмотрим следующий сценарий. Он находит все элементы `<img>` в документе и выполняет их перебор в цикле для поиска тех объектов, которые оказываются (судя по их размеру) рекламными баннерами. Найдя баннер, сценарий при помощи свойства `style.visibility` устанавливает значение атрибута CSS `visibility` равным `hidden`, что делает баннер невидимым:

```
var imgs = document.getElementsByTagName("img");  // Находим все изображения
for(var i = 0; i < imgs.length; i++) {            // Цикл по ним
    var img=imgs[i];
    if (img.width == 468 && img.height == 60)     // Если это баннер 468x60...
        img.style.visibility = "hidden";          // прячем его!
}
```

Я трансформировал этот простой сценарий в «букмарклет» (bookmarklet), преобразовав его в URL `javascript:` и добавив в закладки броузера. Я получаю особое удовольствие, используя эту закладку для моментального скрывания неприятных, постоянно мелькающих анимированных баннеров. Вот версия сценария, пригодная для помещения в закладку:

```
javascript:a=document.getElementsByTagName("img");for(n=0;n<a.length;n++){
i=a[n];if(i.width==468&&i.height==60)i.style.visibility="hidden";}void 0;
```

Код этого «букмарклета» написан очень компактно, с намерением поместить все в одну строку. `javascript:` в начале этого сценария идентифицирует его как URL, телом которого является строка исполняемого содержимого. Предложение `void 0` в конце приводит к тому, что код возвращает неопределенное значение, свидетельствующее, что броузер продолжает отображать текущую веб-страницу (минус, конечно, ее баннеры!). Без `void 0` броузер перепишет текущую веб-страницу с результатом, возвращаемым последней выполненной инструкцией JavaScript.

18.3.1. Соглашения об именах: атрибуты CSS в JavaScript

В именах многих атрибутов стилей CSS, таких как `font-family`, содержатся дефисы. В JavaScript дефис интерпретируется как знак минус, поэтому нельзя написать такое, например, выражение:

```
element.style.font-family = "sans-serif";
```

И поэтому имена свойств объекта CSS2Properties слегка отличаются от имен реальных атрибутов CSS. Если имя атрибута CSS содержит дефисы, имя свойства объекта CSS2Properties образуется путем удаления дефисов и перевода в верхний регистр буквы, непосредственно следующей за каждым из них. Другими словами, атрибут `border-left-width` доступен через свойство

borderLeftWidth, а к атрибуту font-family можно обратиться с помощью следующего кода:

```
element.style.fontFamily = "sans-serif";
```

Есть еще одно отличие в именах атрибутов CSS и свойств объекта CSS2Properties в JavaScript. Слово «float» является ключевым словом в Java и других языках, и хотя сейчас оно не употребляется в JavaScript, оно зарезервировано на будущее. Поэтому в объекте CSS2Properties не может быть свойства с именем float, соответствующего атрибуту CSS float. Затруднение преодолевается путем добавления префикса «css» к атрибуту float, в результате чего образуется имя свойства cssFloat. Следовательно, значение атрибута float элемента можно установить или получить при помощи свойства cssFloat объекта CSS2Properties.

18.3.2. Работа со свойствами стилей

При работе со свойствами стилей объекта CSS2Properties помните, что все значения должны быть указаны в виде строк. В таблице стилей или атрибуте style можно написать:

```
position: absolute; font-family: sans-serif; background-color: #ffffff;
```

Чтобы сделать то же самое для элемента e в JavaScript, необходимо поместить все эти значения в кавычки:

```
e.style.position = "absolute";
e.style.fontFamily = "sans-serif";
e.style.backgroundColor = "#ffffff";
```

Обратите внимание, что точки с запятыми оказываются вне строк. Это обычные точки с запятой, употребляемые в синтаксисе JavaScript. Точки с запятой, используемые в таблицах стилей CSS, не нужны в строковых значениях, устанавливаемых с помощью JavaScript.

Кроме того, помните, что во всех свойствах позиционирования должны быть указаны единицы измерения. Следовательно, нельзя устанавливать свойство left подобным образом:

```
e.style.left = 300;    // Неправильно: это число, а не строка
e.style.left = "300";  // Неправильно: отсутствуют единицы измерения
```

Единицы измерения обязательны при установке свойств стиля в JavaScript – так же как при установке атрибутов стиля в таблицах стилей. Вот правильный способ установки значения свойства left элемента e равным 300 пикселам:

```
e.style.left = "300px";
```

Для того чтобы установить свойство left равным вычисляемому значению, обязательно добавьте единицы измерения в конце вычислений:

```
e.style.left = (x0 + left_margin + left_border + left_padding) + "px";
```

Как побочный эффект добавления единиц измерения, добавление строки преобразует вычисленное значение из числа в строку.

Объект CSS2Properties может также использоваться для получения значений атрибутов CSS, явно установленных в атрибуте `style` элемента, или для чтения любых встроенных значений стилей, ранее установленных кодом JavaScript. Однако и здесь необходимо помнить, что значения, полученные от этих свойств, представляют собой строки, а не числа, поэтому следующий код (предполагающий, что для элемента `e` с помощью встраиваемых стилей установлены поля) не делает того, что от него, возможно, ожидалось:

```
var totalMarginWidth = e.style.marginLeft + e.style.marginRight;
```

А вот такой код будет работать правильно:

```
var totalMarginWidth = parseInt(e.style.marginLeft) + parseInt(e.style.margin-
Right);
```

Это выражение просто отбрасывает спецификации единиц измерения, возвращаемые в конце обеих строк. В нем предполагается, что свойства `marginLeft` и `marginRight` заданы с одинаковыми единицами измерения. Если во встроенных стилях в качестве единицы измерения указаны исключительно пикселы, то, как правило, можно обойтись подобным отбрасыванием единиц измерения.

Вспомните, что некоторые атрибуты CSS, например `margin`, представляют собой сокращения от других свойств, например `margin-top`, `margin-right`, `margin-bottom` и `margin-left`. Объект CSS2Properties имеет свойства, соответствующие этим атрибутам-сокращениям. Так, можно установить свойство `margin` следующим образом:

```
e.style.margin = topMargin + "px " + rightMargin + "px " +
                 bottomMargin + "px " + leftMargin + "px";
```

Возможно, кому-то будет проще установить отдельно четыре свойства полей:

```
e.style.marginTop = topMargin + "px";
e.style.marginRight = rightMargin + "px";
e.style.marginBottom = bottomMargin + "px";
e.style.marginLeft = leftMargin + "px";
```

Можно также получить значения свойств-сокращений, но это редко имеет смысл, поскольку обычно в этом случае приходится разбирать полученное значение на отдельные компоненты. Как правило, это сделать сложно, и намного проще получить свойства-компоненты по отдельности.

И наконец, позвольте мне снова подчеркнуть, что когда вы получаете объект CSS2Properties через свойство `style` объекта HTMLElement, свойства этого объекта представляют значения встроенных атрибутов стилей элемента. Другими словами, установка одного из этих свойств эквивалентна установке атрибута CSS в атрибуте `style` элемента – она влияет только на один данный элемент и имеет приоритет над конфликтующими стилевыми установками

из других источников в каскаде CSS. Именно такое точное управление отдельными элементами и требуется при создании эффектов DHTML с помощью JavaScript.

Однако когда мы читаем значения этих свойств CSS2Properties, они возвращают осмысленные значения, только если были ранее установлены нашим кодом JavaScript или если элемент HTML, с которым мы работаем, имеет встроенный атрибут `style`, установивший нужное свойство. Например, документ может включать таблицу стилей, устанавливающую левое поле для всех абзацев равным 30 пикселам, но если прочитать свойство `leftMargin` одного из элементов-абзацев, будет получена пустая строка, если только этот абзац не имеет атрибута `style`, перекрывающего установку таблицы стилей. Поэтому, несмотря на то что объект CSS2Properties полезен для установки стилей, перекрывающих другие стили, он не дает способа запросить каскад CSS и определить полный набор стилей, применяемых к данному элементу. Далее в этой главе мы кратко рассмотрим метод `getComputedStyle()`, предоставляющий именно такую возможность.

18.3.3. Пример: динамические гистограммы

Графики и диаграммы, добавляемые в HTML-документы, обычно реализуются в виде динамических встраиваемых изображений. Однако поскольку модель компоновки CSS в значительной степени основана на прямоугольных областях, можно динамически создавать гистограммы с помощью JavaScript, HTML и CSS. В примере 18.3 показано, как это сделать. В нем определена функция `makeBarChart()`, упрощающая вставку гистограмм в HTML-документы.

В данном примере для создания новых элементов `<div>` и добавления их в документ применяется технология, рассмотренная в главе 17, а для установки свойств стилей элементов, созданных в этом примере, – методы, обсуждавшиеся в этой главе. Ни текстовое, ни какое-либо другое содержимое не используется, гистограмма – это просто группа прямоугольников, размеры и положение которых аккуратно определены внутри другого прямоугольника. Видимость прямоугольников обеспечивается установкой атрибутов CSS `border` и `background-color`.

Пример включает несложные вычисления высоты каждого прямоугольника в пикселах на основе значений, по которым строится диаграмма. Код JavaScript, устанавливающий положение и размер диаграммы и ее столбцов, также включает простую арифметику для учета границ и отступов. С помощью приемов, показанных в этом примере, можно изменить пример 18.2, написав функцию JavaScript, динамически создающую окна любого заданного размера.

На рис. 18.3 показана диаграмма, созданная с помощью функции `makeBarChart()` следующим образом:

```
<html>
<head>
```

```
<title>Демонстрация BarChart</title>
<script src="BarChart.js"></script>
</head>
<body>
<h1>y = 2<sup>n</sup></h1>
<script>makeBarChart([2,4,8,16,32,64,128,256,512], 600, 250, "red");</script>
<i>Обратите внимание, что каждый столбец в два раза выше, чем
предыдущий. Это результат быстрого экспоненциального роста.</i>
</body>
</html>
```

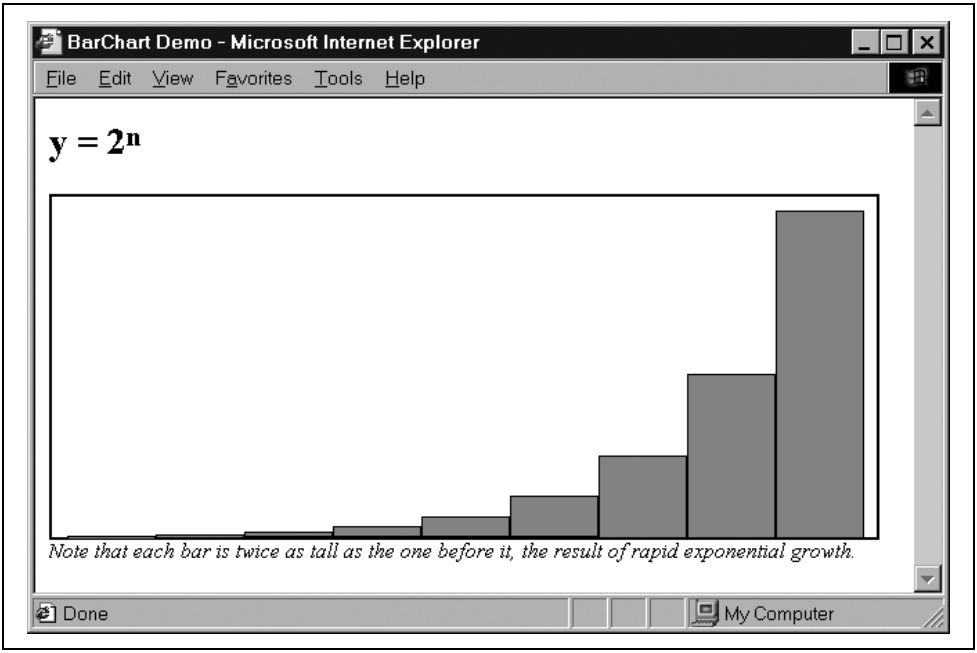

Рис. 18.3. *Динамически созданная гистограмма*

Пример 18.3. Динамическое создание гистограмм

```
/**
 * BarChart.js:
 * В этом файле определяется функция makeBarChart(),  создающая гистограмму
 * для вывода чисел из массива data[]. Гистограмма - это блочный элемент,
 * вставляемый в конец текущего документа. Общий размер гистограммы задается
 * необязательными аргументами width и height, которые включают место,
 * необходимое для границ и внутренних отступов гистограммы.
 * Необязательный аргумент barcolor задает цвет столбцов. Функция
 * возвращает созданный ею элемент chart, чтобы вызывающая сторона могла
 * продолжить работу с ним, установив, например, размер полей.
 *
 * Импортируйте эту функцию в HTML-файл с помощью следующего кода:
 *    <script src="BarChart.js"></script>
```

```
 * Используйте эту функцию в HTML-файле с помощью кода вроде
 *    <script>makeBarChart([1,4,9,16,25], 300, 150, "yellow");</script>
 **/
function makeBarChart(data, width, height, barcolor) {
    // Предоставляет значения по умолчанию для необязательных аргументов
    if (!width) width = 500;
    if (!height) height = 350;
    if (!barcolor) barcolor = "blue";

    // Аргументы width и height задают общий размер генерируемой
    // гистограммы. Мы должны вычесть размеры границ и дополняющих пробелов
    // для получения размера создаваемого элемента.
    width -= 24; // Вычитаем 10px на доп.пробелы и 2px левой и правой границы
    height -= 14;// Вычитаем 10px верхнего доп.пробела и 2px верхней и нижней границ

    // Теперь создаем элемент, в котором будет находиться гистограмма. Обратите
    // внимание, что мы делаем гистограмму относительно позиционированной, чтобы
    // она могла иметь абсолютно позиционированные дочерние элементы и при этом
    // все равно присутствовать в нормальном потоке элементов
    var chart = document.createElement("DIV");
    chart.style.position = "relative";       // Устанавливаем относительное
                                             // позиционирование
    chart.style.width = width + "px";        // Устанавливаем ширину гистограммы
    chart.style.height = height + "px";      // Устанавливаем высоту гистограммы
    chart.style.border = "solid black 2px";  // Задаем для нее границу
    chart.style.paddingLeft = "10px";        // Добавляем пробелы слева,
    chart.style.paddingRight = "10px";       // справа,
    chart.style.paddingTop = "10px";         // и сверху,
    chart.style.paddingBottom = "0px";       // но не снизу
    chart.style.backgroundColor = "white";   // Делаем фон гистограммы белым

    // Вычисляем ширину каждого столбца.
    var barwidth = Math.floor(width/data.length);
    // Находим максимальное число в data[]. Обратите внимание на разумное
    // применение Function.apply().
    var maxdata = Math.max.apply(this, data);
    // Масштабный коэффициент гистограммы: scale*data[i] дает высоту столбца.
    var scale = height/maxdata;

    // Теперь выполняем цикл по массиву данных и создаем столбец
    // для каждого значения.
    for(var i = 0; i < data.length; i++) {
        var bar = document.createElement("div");   // Создаем div для столбца
        var barheight = data[i] * scale;           // Вычисляем высоту столбца
        bar.style.position = "absolute";           // Устанавливаем его положение
                                                   // и размер
        bar.style.left = (barwidth*i+1+10)+"px";   // Прибавляем границу столбца
                                                   // и отступ гистограммы
        bar.style.top = height-barheight+10+"px";  // Добавляем дополнительный
                                                   // пробел гистограммы
        bar.style.width = (barwidth-2) + "px";     // -2 для границы столбца
```

```
        bar.style.height = (barheight-1) + "px";   // -1 для верхней границы столбца
        bar.style.border = "solid black 1px";      // Стиль границы столбца
        bar.style.backgroundColor = barcolor;      // Цвет столбца
        bar.style.fontSize = "1px";                // Обход ошибки в IE
        chart.appendChild(bar);                   // Добавляем столбец к гистограмме
    }

    // Теперь добавляем построенную нами гистограмму в тело документа.
    document.body.appendChild(chart);

    // И наконец, возвращаем элемент chart вызывающей стороне
    // для дальнейших манипуляций.
    return chart;
}
```

18.3.4. Анимация DHTML

Одной из наиболее мощных технологий DHTML, которую можно реализовать с помощью JavaScript и CSS, является анимация. В анимации DHTML нет ничего особенного, – надо лишь периодически изменять одно или несколько свойств стиля одного или нескольких элементов. Для того чтобы, например, передвинуть изображение влево, надо постепенно увеличивать значение свойства `style.left` этого изображения, пока последнее не займет требуемое положение. Или можно постепенно изменять свойство `style.clip` для «открытия» изображения пиксел за пикселом.

Пример 18.4 содержит простой файл HTML, определяющий элемент `div`, который будет анимироваться, и короткий сценарий, изменяющий цвет фона элемента каждые 500 миллисекунд. Обратите внимание, что изменение цвета выполняется присваиванием значения свойства стиля CSS. Анимация возникает за счет того, что изменение цвета выполняется периодически, с помощью функции `setInterval()` объекта Window. (Никакая анимация в динамическом HTML не обходится без метода `setInterval()` или `setTimeout()`; возможно, вы захотите освежить эти методы в памяти, прочитав соответствующие разделы в «Справочнике по клиентскому JavaScript».) И наконец, обратите внимание на применение оператора «деление по модулю» (получение остатка) `%` для перебора цветов. Тот, кто забыл, как работает этот оператор, может обратиться к главе 5.

Пример 18.4. Простая анимация «изменение цвета»

```
<!-- Этот div - это элемент, с которым будет выполняться анимация -->
<div id="urgent"><h1>Внимание!</h1>Веб-сервер атакован!</div>
<!-- Это сценарий анимации элемента -->
<script>
var e = document.getElementById("urgent");          // Получаем объект Element
var colors = ["white", "yellow", "orange", "red"]  // Перебираемые цвета
var nextColor = 0;                                  // Текущая позиция перебора
// Для анимации с цветом фона элемента div вычисляем следующее
// выражение каждые 500 миллисекунд.
setInterval("e.style.backgroundColor=colors[nextColor++%colors.length];", 500);
</script>
```

Пример 18.4 реализует очень простую анимацию. На практике анимации с использованием CSS обычно включают одновременную модификацию нескольких свойств стилей (таких как top, left и clip). Сложную анимацию с помощью технологии, показанной в примере 18.4, создать трудно. Кроме того, чтобы не надоедать пользователю, анимация должна выполняться в течение короткого периода времени и затем останавливаться (для анимации, создаваемой примером 18.4, способ остановки не реализован).

В примере 18.5 показан файл JavaScript, определяющий функцию анимации CSS, намного облегчающей эту задачу, даже при создании сложной анимации. Функции animateCSS(), определенной в этом примере, передается пять аргументов. Первый аргумент задает анимируемый объект HTMLElement. Второй и третий аргументы задают количество кадров анимации и продолжительность времени, в течение которого должен быть показан каждый кадр. Четвертый аргумент – это объект JavaScript, определяющий выполняемую анимацию. А пятый аргумент – это необязательная функция, которая должна быть вызвана один раз по завершении анимации.

Ключевым является четвертый аргумент функции animateCSS(). Каждое свойство объекта JavaScript должно иметь то же имя, что и свойство стиля в CSS, а значением каждого свойства должна быть функция, возвращающая допустимое значение для стиля с этим именем. При отображении нового кадра вызывается каждая из этих функций, чтобы сгенерировать новые значения для каждого атрибута стиля. Каждой функции передается номер кадра и общее время, прошедшее с начала анимации, и функция может использовать эти аргументы для вычисления нужного значения.

Пример должен в значительной мере прояснить применение animateCSS(). Следующий код перемещает элемент вверх по экрану, при этом постепенно открывая его путем увеличения области отсечения.

```
// Анимируем элемент с id "title" 40 кадрами по 50 миллисекунд каждый
animateCSS(document.getElementById("title"), 40, 50,
       {  // Так устанавливаются стилевые свойства top и clip для каждого кадра:
            top:  function(f,t) { return 300-f*5 + "px"; }
            clip: function(f,t) {return "rect(auto "+f*10+"px auto auto)";},
          });
```

Следующий фрагмент кода с помощью animateCSS() перемещает кнопку Button по кругу. Необязательный пятый аргумент передается функции для изменения текста кнопки на «Готово», когда анимация завершена. Функции, указанной пятым аргументом, в качестве аргумента передается анимируемый элемент:

```
// Перемещаем кнопку по кругу, а затем изменяем отображаемый ею текст
animateCSS(document.forms[0].elements[0], 40, 50,  // Кнопка, 40 кадров, 50 мс
           {  // Эта тригонометрия определяет круг радиусом 100 и с центром
              // в точке (200,200):
               left: function(f,t){ return 200 + 100*Math.cos(f/8) + "px"},
               top:  function(f,t){ return 200 + 100*Math.sin(f/8) + "px"}
           },
function(button) { button.value = "Кнопка"; });
```

Код примера 18.5 достаточно прост; как мы скоро увидим, вся сложность заключена в свойствах объекта анимации, которые передаются функции `animateCSS()`. Функция `animateCSS()` определяет вложенную функцию под названием `displayNextFrame()` и почти ничего не делает, разве что с помощью `setInterval()` настраивает периодический вызов `displayNextFrame()`. `displayNextFrame()` выполняет цикл по свойствам объекта анимации и вызывает различные функции для вычисления новых значений свойств стиля.

Обратите внимание, что поскольку `displayNextFrame()` определена внутри `animateCSS()`, она имеет доступ к аргументам и локальным переменным `animateCSS()`, даже несмотря на то, что `displayNextFrame()` вызывается уже после завершения функции `animateCSS()`! И все получается, даже если `animateCSS()` вызывается более одного раза для одновременной анимации нескольких элементов. (Если вы не понимаете, почему это работает, можете вернуться к разделу 11.4 «Лексическая область действия и вложенные функции».)

Пример 18.5. Каркас для анимации на базе CSS

```
/**
 * AnimateCSS.js:
 * Этот файл определяет функцию под именем animateCSS(), служающую
 * каркасом для создания анимаций на базе CSS. Аргументы этой функции:
 *
 *     element: Анимируемый HTML-элемент.
 *     numFrames: Общее число кадров в анимации.
 *     timePerFrame: Количество миллисекунд для отображения каждого кадра.
 *     animation: Объект, определяющий анимацию; описан ниже.
 *     whendone: Необязательная функция, вызываемая, когда анимация завершена.
 *               Если эта функция указана, ей в качестве аргумента передается
 *               значение аргумента element.
 *
 * Функция animateCSS() просто определяет каркас для анимации. Выполняемую
 * анимацию определяют свойства объекта animation. Каждое свойство должно
 * иметь то же имя, что и стилевое свойство CSS. Значением каждого свойства
 * должна быть функция, возвращающая значения для этого стилевого свойства.
 * Каждой функции передается номер кадра и общий промежуток времени, прошедший
 * с начала анимации, а функция может использовать это для вычисления
 * стилевого значения, которое она должна вернуть для данного фрейма. Например,
 * чтобы анимировать изображение так, чтобы оно передвигалось из левого
 * верхнего угла, вы можете вызвать animateCSS следующим образом:
 *
 *  animateCSS(image, 25, 50,  // Анимируем изображение в течение 25 кадров
 *                             // по 50 мс каждый
 *             {  // Устанавливаем атрибуты top и left для каждого кадра, как:
 *               top: function(frame,time) { return frame*8 + "px"; },
 *               left: function(frame,time) { return frame*8 + "px"; }
 *             });
 *
 **/
function animateCSS(element, numFrames, timePerFrame, animation, whendone) {
    var frame = 0;  // Здесь хранится текущий номер кадра
```

```
    var time = 0;   // Здесь хранится общее время анимации

    // Настраиваем вызов displayNextFrame() каждые timePerFrame миллисекунд.
    // Так будет отображаться каждый кадр анимации.
    var intervalId = setInterval(displayNextFrame, timePerFrame);

    // На этом вызов animateCSS() завершается, но приведенная выше строка
    // гарантирует, что определенная ниже вложенная функция будет вызываться
    // для каждого кадра анимации. Так как эта функция определена внутри
    // animateCSS(), она имеет доступ к аргументам и локальным переменным
    // animateCSS(), даже несмотря на то, что вызывается после завершения этой
    // функции!
    function displayNextFrame() {
        if (frame >= numFrames) {             // Сначала смотрим, не закончилась
                                              // ли анимация
            clearInterval(intervalId);        // Если да, прекращаем вызовы
            if (whendone) whendone(element);  // Вызываем функцию whendone
            return;                           // Мы закончили
        }

        // Выполняем цикл по всем свойствам, определенным в объекте animation
        for(var cssprop in animation) {
            // Для каждого свойства вызываем его функцию анимации, передавая
            // ей номер кадра и прошедшее время. Используем возвращаемое
            // функцией значение в качестве нового значения соответствующего
            // стилевого свойства указанного элемента. Используем try/catch
            // для игнорирования любых исключительных ситуаций, возникших
            // из-за неверных возвращаемых значений.
            try {
                element.style[cssprop] = animation[cssprop](frame, time);
            } catch(e) {}
        }

        frame++;                // Увеличиваем номер кадра
        time += timePerFrame;   // Увеличиваем прошедшее время
    }
}
```

18.4. DHTML в броузерах четвертого поколения

Именно в броузерах Internet Explorer 4 и Netscape 4 была впервые реализована технология DHTML в Интернете. Оба броузера частично поддерживают стандарт CSS1 и поддерживают атрибуты позиционирования CSS (которые были интегрированы в стандарт CSS2), являющиеся ключевыми для DHTML. К сожалению, во время разработки этих броузеров четвертого поколения стандарт DOM еще не существовал, поэтому они данному стандарту не соответствуют. Тем не менее в обоих броузерах можно добиться DHTML-эффектов.

18.4.1. DHTML в Internet Explorer 4

Как мы видели в главе 17, IE 4 не поддерживает метод `document.getElementById()` и не поддерживает API для динамического создания новых узлов и вставки их в документ. Вместо этого он предоставляет массив `document.all[]` как способ нахождения произвольных элементов в документе и позволяет изменять содержимое документа с помощью свойства `innerHTML` элементов документа. IE 4 в этом не соответствует стандартам, но предоставляет адекватные альтернативы.

Обход и изменение документов – важная часть DHTML, но тема этой главы – динамическое применение стилей CSS. Хорошая новость: DOM API, рассмотренный выше при описании установки атрибутов стилей в CSS через свойство `style`, был взят из API IE 4. Поэтому, обратившись к `document.all[]` для нахождения элемента документа, который вы хотите модифицировать, можно использовать стили этого элемента в сценариях так же, как это делается в броузере, полностью поддерживающем DOM API. (Однако помните, что IE 4 не поддерживает CSS целиком, поэтому не надо ожидать, что из сценариев будут доступны все свойства стилей.)

Стандарт CSS2 определяет, что атрибут `position` может применяться для задания абсолютного или относительного позиционирования любого элемента в документе. Однако IE 4 был реализован до завершения CSS2 и поддерживает абсолютное позиционирование только для определенного подмножества элементов. Следовательно, при задании абсолютного позиционирования в IE 4 необходимо заключать содержимое, которое будет позиционировано или анимировано, в теги `<div>` или `<span>`, у которых есть атрибут `position`.

18.4.2. DHTML в Netscape 4

Создание эффектов DHTML с помощью Netscape 4 – дело более сложное. Netscape 4 не поддерживает полную объектную модель и поэтому не позволяет программам на JavaScript обращаться к произвольным элементам HTML. Следовательно, Netscape 4 не может обеспечить доступ к встраиваемым стилям произвольных элементов. Вместо этого он определяет специальный объект Layer[1]. Любой абсолютно позиционированный элемент (то есть любой элемент, у которого атрибут `position` установлен в `absolute`) помещается на отдельный от остальной части документа слой. Этот слой может независимо позиционироваться, скрываться, показываться, опускаться ниже или подниматься выше остальной части документа и т. д. Layer API был представлен для стандартизации в W3C, но никогда не был стандартизован. По этой причине он был исключен из проекта Mozilla и не поддерживается ни в Mo-

[1] Слои были введены в главе 17, когда мы обсуждали несовместимость Netscape 4 с базовым DOM API. Здесь мы продолжим это обсуждение и узнаем, каким образом слои предоставляют альтернативу базовому DOM API и DOM API для доступа к стилям CSS.

zilla, ни в Netscape 6. Следовательно, приемы, описанные в этом разделе, действуют только в броузерах версии 4.x от Netscape.

Каждый независимо позиционированный слой в документе представлен объектом Layer, и, что неудивительно, массив `layers[]` объекта Document содержит полный набор объектов Layer в документе. (Объекты Layer расположены в этом массиве в том же порядке, что и в документе.) Кроме того, к любому слою, которому с помощью атрибута `name` или `id` присвоено имя, можно по этому имени получить доступ. Например, если для слоя указано `id="L2"`, можно обращаться к нему в Netscape как к `document.L2` или как к `document.layers["L2"]`. Несмотря на то что Netscape 4 не предоставляет способа обращаться к произвольным элементам документа, этот массив `layers[]` обеспечивает доступ к наиболее важным динамическим элементам.

Слой – это что-то вроде отдельного окна или фрейма. Хотя объект Layer – это не то же самое, что и объект Window, – он, как окна и фреймы, имеет свойство `document`. Свойство `document` объекта Layer ссылается на объект Document: в каждом слое есть свой собственный совершенно независимый документ HTML. Слои даже могут быть вложенными; текст HTML можно выводить во вложенный слой с помощью следующего кода:

```
document.layers[1].document.layers[0].document.write("Слои - это интересно!");
document.layers[1].document.layers[0].document.close();
```

Netscape 4 не позволяет создавать узлы дерева документа или манипулировать ими и даже не поддерживает свойство `innerHTML` из Internet Explorer. Однако тот факт, что слои содержат независимые документы, обеспечивает способ динамической модификации содержимого документа.

Несмотря на то что Netscape 4 определяет слой как элемент с установленным стилем CSS `position`, он не определяет какого-либо способа непосредственно обращаться к стилям элемента слоя из сценариев. Вместо этого объект Layer определяет свойства и методы, с помощью которых можно динамически позиционировать слои.

Имена свойств объекта Layer похожи на имена важных атрибутов стилей в CSS, но эти свойства не являются полным аналогом свойств стилей. Например, свойства `left` и `top` объекта Layer задают позицию слоя в пикселах; установка этих свойств слоя аналогична установке стилевых свойств `left` и `top` элемента, за исключением того, что свойства объекта Layer принимают числовые значения (количество пикселов), а не строки, включающие числовое значение и спецификацию единицы измерения. Свойство `visibility` для слоя задает, должно ли содержимое слоя быть видимым; оно во многом схоже со свойством стиля с тем же именем, за исключением того, что принимает значения `show` или `hide` вместо положенных по стандарту CSS `visible` и `hidden`. Объект Layer также поддерживает свойство `zIndex`, работающее аналогично свойству стиля `zIndex`. В табл. 18.3 перечислены ключевые свойства стилей CSS и свойства объекта Layer, наиболее близко соответствующие каждому из них. Обратите внимание, что это все свойства стилей, доступные из сценариев в Netscape 4.

Таблица 18.3. Свойства объекта Layer в Netscape 4

Свойство CSS	Эквивалентное свойство Layer	Примечания об объекте Layer
left, top	left, top	Задает количество пикселов, без указания единиц измерения. См. также moveTo() и moveBy().
zIndex	zIndex	См. также moveAbove(), moveBelow().
visibility	visibility	Layer возвращает show или hide, даже если это свойство установлено равным стандартным значениям visible и hidden.
clip	clip.bottom, clip.height, clip.left, clip.right, clip.top, clip.width	Задает пикселы без единиц измерения.
backgroundColor	bgColor	
backgroundImage	background.src	Устанавливает строку URL.

Как можно видеть из табл. 18.3, объект Layer поддерживает пару полезных свойств, не относящихся к динамическому позиционированию. Свойство background.src задает фоновое изображение для слоя, а свойство bgColor — цвет фона для слоя. Эти свойства соответствуют свойствам стилей backgroundImage и backgroundColor.

В дополнение к своим свойствам объект Layer предлагает несколько удобных методов. Методы moveBy() и moveTo() перемещают слой на относительное смещение или в абсолютную позицию. Методы moveAbove() и moveBelow() устанавливают свойство zIndex для слоя относительно некоторого другого слоя. Полный список свойств и методов объекта Layer приведен в «Справочнике по клиентскому JavaScript» в этой книге.

Каждый слой содержит независимый документ, поэтому можно динамически обновлять содержимое слоя с помощью методов open(), write() и close() объекта Document, как мы видели в главе 14. Кроме того, свойство src слоя задает URL документа, который он отображает. Установив это свойство, можно заставить броузер загрузить для отображения в слое совершенно новый документ. Аналогичный эффект производит метод load(); он загружает новый URL и одновременно изменяет ширину слоя. Поскольку слои часто содержат динамически генерируемое содержимое, кому-то будет удобно использовать со свойством src и методом load() URL javascript:.

Мы видели, что Netscape 4 автоматически создает объект Layer для любого элемента, у которого значение свойства стиля position равно absolute. API Netscape 4 также позволяет создавать слои другими, менее стандартными способами. Например, Netscape 4 определяет тег HTML <layer>, позволяющий определять слои непосредственно в коде HTML. Тег <layer> остается фирменным расширением Netscape 4, он не был включен в стандарт HTML 4 и не поддерживается в Mozilla и Netscape 6. Однако важнее, что Netscape 4 поддерживает конструктор Layer(), позволяющий при необходимости дина-

мически создавать объекты Layer внутри программы. Подробности приведены в «Справочнике по клиентскому JavaScript» в этой книге.

18.4.3. Пример: кросс-платформенная анимация DHTML

Несмотря на различия между DOM API, IE 4 API и Netscape Layer API, можно создать эффекты DHTML, работающие в совместимых с DOM броузерах, в версиях IE до DOM и в Netscape 4. Пример 18.6 показывает один из способов, позволяющих это сделать. Этот сценарий отображает слово «Hello» и анимирует его, перемещая его по прямой от одной точки в окне броузера к другой.

Обратите внимание на технологию обеспечения совместимости, реализованную в этом примере: мы проверяем существование ключевых функций, массивов и свойств перед обращением к ним. Если в объекте Document имеется свойство с именем `getElementById`, мы предполагаем, что имеется броузер, совместимый с DOM, и это свойство ссылается на метод `getElementById()`. Аналогично, если объект Document имеет свойство с именем `all`, мы предполагаем, что сценарий запущен в Internet Explorer, и используем для нахождения анимируемого элемента массив `document.all[]`.

Пример 18.6. Кросс-платформенный сценарий анимации DHTML

```
<!-- Это динамический элемент, который мы будем анимировать. Мы заключаем -->
<!-- тег h1 в div, так как IE 4 не перемещает h1 без наличия контейнера   -->
<!-- div или span                                                         -->
<div id="title" style="position:absolute"><h1>Hello</h1></div>

<!-- Это код JavaScript, осуществляющий анимацию -->
<script>
// Эти переменные устанавливают параметры для нашей анимации:
var id = "title";                  // Имя анимируемого элемента
var numFrames = 30;                // Сколько отображается кадров
var interval = 100;                // Как долго отображается каждый кадр
var x0 = 100, y0 = 100;            // Начальная позиция элемента
var x1 = 500, y1 = 500;            // Конечная позиция элемента
var dx = (x1 - x0)/(numFrames-1);  // Расстояние горизонтального перемещения в
                                   // каждом кадре
var dy = (y1 - y0)/(numFrames-1);  // Расстояние вертикального перемещения в
                                   // каждом кадре
var frameNum = 0;                  // Текущий кадр
var element = null;                // Анимируемый элемент

// Сначала находим анимируемый элемент. Если броузер поддерживает,
// используем технологию, совместимую с DOM; в противном случае возвращаемся
// к специфическому для броузера коду.
if (document.getElementById) {              // Если это броузер, совместимый с DOM,
    element = document.getElementById(id);  // используем метод DOM
}
else if (document.all) {            // В противном случае, если поддерживается
    element = document.all[id];     // IE API, используем для нахождения
```

```
}                                   // элемента массив all[]
else if (document.layers) {         // Иначе, если поддерживается Netscape API,
    element = document.layers[id];  // для получения элемента используем массив
}                                   // layers[]

// Если мы нашли анимируемый элемент одним из приведенных выше способов,
// начинаем анимацию путем вызова nextFrame() каждые interval миллисекунд
if (element) {
    var intervalId = setInterval("nextFrame()", interval);
}

// Эта функция периодически вызывается для отображения каждого кадра анимации.
// Она перемещает элемент, используя либо DOM API для установки стилевых свойств,
// либо, если броузер не поддерживает этот API, Netscape Layer API.
function nextFrame() {
    if (element.style) {
        // Если броузер это поддерживает, перемещаем элемент путем установки
        // стилевых свойств CSS. Обратите внимание на добавление единиц измерения.
        element.style.left = x0 + dx*frameNum + "px";
        element.style.top = y0 + dy*frameNum + "px";
    }
    else {
        // В противном случае предполагаем, что элемент является слоем,
        // и перемещаем его, устанавливая свойства слоя. Мы также могли бы
        // использовать метод element.moveTo().
        element.left = x0 + dx*frameNum;
        element.top = y0 + dy*frameNum;
    }

    // Увеличиваем номер кадра и останавливаемся, если достигли конца анимации.
    if (++frameNum >= numFrames) clearInterval(intervalId);
}
</script>
```

18.5. Другие DOM API для стилей и таблиц стилей

В этой главе мы обсуждали простой DOM API для работы со стилями CSS: каждый объект HTMLElement в документе имеет свойство `style`, представляющее встроенные атрибуты стилей данного элемента. Свойство `style` ссылается на объект CSS2Properties, в котором определяется свойство JavaScript для каждого атрибута стиля CSS, определенного в стандарте CSS2.

Несмотря на то что мы активно использовали объект CSS2Properties, он представляет собой только одну часть DOM API для CSS.[1] Этот раздел содержит краткий обзор остальной части DOM API для работы с таблицами сти-

[1] На самом деле объект CSS2Properties не является обязательным. Реализация DOM может поддерживать CSS, не поддерживая CSS2Properties. Однако на практике этот API наиболее часто применяется для работы со стилями, и, по существу, для реализаций DOM в веб-броузерах его поддержка стала обязательной.

лей CSS. Однако обратите внимание, что на момент написания этой книги значительная часть CSS API не так хорошо поддерживалась броузерами шестого поколения. Прежде чем полагаться на какой-либо из описанных здесь API, следует тщательно протестировать его поддержку.

18.5.1. Объявления стилей

Интерфейс CSS2Properties – это подынтерфейс CSSStyleDeclaration. Следовательно, свойство `style` каждого элемента документа также реализует свойства и методы CSSStyleDeclaration. Среди этих методов есть `setProperty()` и `getPropertyValue()`, которые можно использовать в качестве альтернативы установке и получению отдельных свойств стилей CSS2Properties. Например, следующие две строки кода выполняют одно и то же действие:

```
element.style.fontFamily = "sans-serif";
element.style.setProperty("font-family", "sans-serif", "");
```

Другие возможности интерфейса CSSStyleDeclaration включают метод `removeProperty()`, удаляющий указанный стиль, и свойство `cssText`, возвращающее текстовое представление всех атрибутов стилей и их значений. Так как объекты CSSStyleDeclaration представляют набор атрибутов стилей и их значений, они также могут выступать в качестве массивов для выполнения цикла по именам атрибутов стилей.

18.5.2. Вычисляемые стили

Как я подчеркивал ранее в этой главе, свойство `style` элемента документа представляет атрибут `style` для этого элемента и не содержит какой-либо информации о других стилях (из таблицы стилей), влияющих на этот элемент. Чтобы определить полный набор стилей, применяемых к элементу, используйте метод `getComputedStyle()` объекта Window (этот метод определяется интерфейсом AbstractView: см. раздел «AbstractView, getComputedStyle()» в «Справочнике по W3C DOM»). Этот метод возвращает значение, представляющее собой объект CSSStyleDeclaration, описывающий все стили, применяемые к указанному элементу. Можно считать, что возвращаемый объект также реализует интерфейс CSS2Properties, как и свойство `style`.

Чтобы проиллюстрировать различие между встроенным стилем элемента и его вычисляемым стилем, рассмотрим элемент `e`. Определить, указан ли шрифт во встраиваемом атрибуте `style` элемента `e`, можно так:

```
var inlinefont = e.style.fontFamily;
```

Но чтобы определить, какой шрифт используется для отображения `e` (независимо от того, указано ли это во встроенном стиле или в таблице стилей), надо сделать следующее:

```
var fontfamily = window.getComputedStyle(e, null).fontFamily;
```

Можно предпочесть getComputedStyle(), чтобы было понятно, что этот метод определяется интерфейсом AbstractView:

```
var fontfamily = document.defaultView.getComputedStyle(e, null).fontFamily;
```

Значения стилей, возвращаемые getComputedStyle(), доступны только для чтения, так как они поступают из различных мест в каскаде стилей. Установка любого атрибута не влияет на стиль элемента. Кроме того, метод getComputedStyle() следует считать «дорогостоящим», поскольку он должен обойти весь каскад и сформировать большой объект CSSStyleDeclaration, представляющий много атрибутов стилей, которые применяются к элементу.

И наконец, обратите внимание, что IE 5 и более поздние версии в дополнение к свойству style определяют для всех элементов HTML нестандартное, но полезное свойство currentStyle. currentStyle ссылается на объект CSS2Properties, содержащий вычисляемый стиль данного элемента.

18.5.3. Перекрывающие стили

Стандарт CSS указывает, что у веб-броузера есть таблица стилей по умолчанию, определяющая базовые стили отображения элементов HTML. Броузер может разрешать пользователю указывать пользовательскую таблицу стилей, представляющую его стилевые предпочтения и перекрывающую стили, определенные в таблице стилей по умолчанию. Авторские таблицы стилей – это таблицы стилей, определенные автором документа, то есть стили, включенные в документ или на которые есть ссылка в документе. Авторские таблицы стилей перекрывают стили по умолчанию броузера и пользовательские стили (за исключением стилей !important). Встроенные стили, указанные с помощью атрибута style элемента, могут рассматриваться как часть авторской таблицы стилей.

Стандарт DOM вводит понятие перекрывающей таблицы стилей, которая перекрывает авторскую таблицу стилей, включая встроенные стили. Установив стили для элемента в перекрывающей таблице стилей, можно изменить стиль отображения элемента, не изменяя таблицу стилей элемента или его встроенный стиль. Перекрывающий стиль для элемента можно получить с помощью метода getOverrideStyle() объекта Document:

```
var element = document.getElementById("title");
var override = document.getOverrideStyle(element, null);
```

Этот метод возвращает объект CSSStyleDeclaration (также реализующий CSS2Properties), с помощью которого можно изменить стиль отображаемого элемента. Обратите внимание на различие между установкой перекрывающего стиля и встраиваемого стиля:

```
override.backgroundColor = "yellow";      // Устанавливает перекрывающий стиль
element.style.backgroundColor = "pink";   // Устанавливает встраиваемый стиль
```

18.5.4. Создание таблиц стилей

Объект DOMImplementation (доступный как `document.implementation`) определяет метод `createCSSStyleSheet()` для создания объектов CSSStyleSheet. Объект CSSStyleSheet определяет метод `insertRule()`, позволяющий добавлять правила в таблицу стилей. К сожалению, DOM Level не определяет какого-либо способа связи созданной таблицы стилей с документом, поэтому пока нет смысла использовать этот метод. В будущих версиях стандарта DOM это может быть исправлено.

18.5.5. Обход таблицы стилей

Базовый DOM API позволяет обойти документ HTML (или XML) и просмотреть каждый элемент, атрибут или текстовый узел документа. Аналогичным образом, таблицы стилей и модули CSS в DOM делают возможным просмотр всех встроенных и связанных с документом таблиц стилей и позволяют обойти эти таблицы стилей, просмотрев все составляющие их правила, селекторы и атрибуты стилей.

Авторам сценариев, имеющим дело с DHTML, как правило, достаточно работать со встроенными стилями элементов с помощью API, показанного ранее в этой главе, и зачастую обход таблиц стилей не требуется. Тем не менее в этой главе есть краткое введение в DOM API для обхода таблиц стилей. Подробную информацию по этому API можно найти в «Справочнике по W3C DOM». На момент написания этой книги данный API поддерживался не очень хорошо, но скоро ожидается появление этой поддержки в Mozilla. Заметим также, что IE 5 определяет собственный и несовместимый API для обхода таблиц стилей.

Доступ к таблицам стилей, встроенным и связанным с документом, можно получить через массив `document.styleSheets[]`. Например:

```
var ss = document.styleSheets[0];
```

Элементами этого массива являются объекты StyleSheets, представляющие обобщенную таблицу стилей. В HTML-документах, использующих таблицы стилей CSS, все эти объекты реализуют подынтерфейс CSSStyleSheet, предоставляющий свойства и методы, специфические для CSS. В объекте CSSStyleSheet имеется массив `cssRules[]`, содержащий правила таблицы стилей:

```
var firstRule = document.styleSheets[0].cssRules[0]
```

Интерфейс CSSStyleSheet также определяет методы `insertRule()` и `deleteRule()` для добавления и удаления правил из таблицы стилей.

```
document.styleSheets[0].insertRule("H1 { text-weight: bold; }", 0);
```

Элементы массива `CSSStyleSheet.cssRules[]` — это объекты CSSRule. Таблицы стилей CSS могут содержать множество различных типов правил. Кроме основных стилевых правил, которые мы видели в этой главе, есть различ-

ные «@-правила», задаваемые с помощью ключевых слов `@import` и `@page`. Об этих специальных типах правил CSS рассказывается в литературе по CSS.

Интерфейс CSSRule – это обобщенный интерфейс, который может представлять любой тип правил и имеет подынтерфейсы для каждого специфического типа правил. Свойство `type` CSSRule задает конкретный тип правила. Большинство правил в таблице стилей CSS являются основными, например:

```
h1 { font-family: sans-serif; font-weight: bold; font-size: 24pt; }
```

Для правил этого типа значение свойства `type` равно `CSSRule.STYLE_RULE`, и они представлены объектами CSSRule, реализующими интерфейс CSSStyleRule. Объекты CSSStyleRule определяют свойство `selectorText`, содержащее селектор правила (в предыдущем примере это «h1»), и свойство `style`, содержащее атрибуты стиля и их значения для данного правила (такие как атрибуты шрифта в предыдущем правиле). Например:

```
var rule = document.styleSheets[0].cssRules[0]
var styles;
if (rule.type == CSSRule.STYLE_RULE) styles = rule.style;
```

Значение свойства `CSSStyleRule.style` – это объект CSSStyleDeclaration. Мы уже встречали этот объект: это тот же самый тип объекта, который используется для представления встроенных стилей элементов документа. Он определяет такие методы, как `setProperty()`, `removeProperty()` и `getPropertyValue()`. Как уже говорилось, объекты CSSStyleDeclaration обычно также реализуют интерфейс CSS2Properties, и, следовательно, определяют свойства, соответствующие каждому атрибуту CSS.

Свойства CSS2Properties и метод `getPropertyValue()` объекта CSSStyleDeclaration возвращают значения атрибутов стилей CSS в виде строк. Как мы уже говорили в данной главе, это значит, что при запросе значения такого атрибута, как `font-size` (или при чтении свойства `fontSize` объекта CSS2Properties), вы получаете число и значение единицы измерения. Это может быть «24pt» или (возможно, менее полезное) значение типа «10mm». Как правило, получив значение атрибута CSS в виде строки, необходимо каким-либо образом разобрать строку, чтобы извлечь из нее нужные данные. Это особенно касается таких атрибутов, как `clip`, имеющих сложный строковый синтаксис.

В качестве альтернативы разбору строк объект CSSStyleDeclaration предоставляет другой метод, `getPropertyCSSValue()`, возвращающий значение атрибута CSS в виде объекта CSSValue вместо строки. Свойство `cssValueType` объекта CSSValue задает подынтерфейс, который также реализуется этим объектом. Если в атрибуте имеется более одного значения, то объект CSSValue реализует CSSValueList и ведет себя как массив объектов CSSValue. В противном случае объект CSSValue обычно является «элементарным» значением и реализует интерфейс CSSPrimitiveValue. В объектах CSSPrimitiveValue есть свойство `primitiveType`, задающее тип значения или единицы измерения, применяемые к этому значению. Имеется 26 возможных типов, пред-

ставленных такими константами, как `CSSPrimitiveValue.CSS_PERCENTAGE`, `CSSPrimitiveValue.CSS_PX` и `CSSPrimitiveValue.CSS_RGBCOLOR`. Кроме свойства `primitiveType` и различных констант типов, CSSPrimitiveValue определяет разные методы для установки и получения значения, представленного объектом. Если объект CSSPrimitiveValue представляет, например, длину или проценты, для получения длины вызывается метод `getFloatValue()`. Если свойство `primitiveType` указывает на то, что значение представляет собой цвет, для запроса значения цвета используется метод `getRGBColorValue()`.

И наконец, DOM CSS API также определяет несколько специальных объектных типов для представления значений атрибутов: объекты RGBColor представляют значения цвета, объекты Rect – значения параметров прямоугольника (например, для атрибута `clip`), а объекты Counter – счетчики CSS2. Подробности можно найти в «Справочнике по W3C DOM».

19

События и обработка событий

Как мы видели в главе 12, интерактивные программы на JavaScript основаны на модели программирования, управляемого событиями. При таком стиле программирования веб-броузер генерирует *событие*, когда с документом или некоторым его элементом происходит что-то интересное. Например, веб-броузер генерирует событие, когда он завершает загрузку документа, когда пользователь перемещает курсор мыши над гиперссылкой или щелкает по кнопке Submit формы. Если приложение JavaScript интересует определенный тип события определенного элемента документа, оно может зарегистрировать *обработчик события* – функцию JavaScript или фрагмент кода – для этого типа события в интересующем вас элементе. Потом, когда возникает это событие, броузер вызывает код обработчика. Все приложения с графическим интерфейсом пользователя разработаны подобным образом: они ждут, пока пользователь что-нибудь сделает (то есть ждут, когда произойдут события), и затем реагируют на действия.

В качестве отступления следует отметить, что таймеры и обработчики ошибок (см. главу 13) связаны с управляемым событиями программированием. Работа таймеров и обработчиков ошибок, как и обработчиков событий, описанных в этой главе, основана на регистрации функции в броузере и последующем вызове броузером этой функции, когда происходит соответствующее событие. Однако в этом случае событием является истечение указанного промежутка времени или возникновение ошибки JavaScript. Хотя таймеры и обработчики ошибок в данной главе не обсуждаются, их полезно рассматривать как относящиеся к обработке событий, и я рекомендую вам заново прочитать раздел 13.4 «Время запуска и интервалы» и раздел 13.5 «Обработка ошибок», переосмыслив их в контексте этой главы.

Обработчики событий активно применяются в наименее тривиальных программах на JavaScript. Некоторые примеры JavaScript с простыми обработ-

чиками событий мы уже видели. Данная глава заполняет подробностями все пробелы в рассмотрении событий и их обработки. К сожалению, эти подробности сложнее, чем должны были быть, поскольку дальнейшее обсуждение основано на четырех различных и несовместимых моделях обработки событий. Вот эти модели.

Исходная событийная модель

Эта простая модель используется (но не документирована тщательно) в данной книге. В ограниченном объеме она была кодифицирована стандартом HTML 4 и неформально рассматривается как часть DOM Level 0 API. Несмотря на ограниченность ее возможностей она поддерживается всеми веб-броузерами, поддерживающими JavaScript, и потому переносима.

Стандартная событийная модель

Это мощная и богатая возможностями модель была стандартизована в DOM Level 2. Поддерживается броузерами Netscape 6 и Mozilla.

Событийная модель Internet Explorer

Эта модель реализована в IE 4 и более поздних версиях и обладает некоторыми, но не всеми возможностями стандартной событийной модели. Несмотря на то что Microsoft участвовала в создании событийной модели DOM Level 2 и располагала достаточным временем для реализации стандартной событийной модели в IE 5.5 и IE 6, разработчики этих броузеров продолжают придерживаться своей фирменной событийной модели. Это значит, что программисты на JavaScript должны писать для броузеров IE специальный код, если хотят получить доступ к развитым возможностям обработки событий.

Событийная модель Netscape 4

Данная модель была реализована в Netscape 4 и продолжает (в основном, но не полностью) поддерживаться в Netscape 6, хотя и была вытеснена стандартной событийной моделью. JavaScript-программисты, интересующиеся развитыми возможностями обработки событий и стремящиеся сохранить совместимость с Netscape 4, должны понимать эту модель.

Все эти событийные модели по очереди будут описаны в данной главе.

19.1. Базовая обработка событий

В примерах, рассмотренных ранее, обработчики событий записывались в виде строк кода JavaScript, выступающих в качестве значений определенных атрибутов HTML, таких как `onclick`. Это основа исходной событийной модели, но есть некоторые дополнительные нюансы, требующие понимания и рассмотренные в следующих разделах.

19.1.1. События и типы событий

Различные типы происшествий генерируют различные типы событий. Перемещая мышь над гиперссылкой и щелкая по кнопке мыши на кнопке Submit формы, пользователь вызывает события разных типов. Даже одно и то же происшествие может генерировать различные типы событий в зависимости от контекста: например, когда пользователь нажимает кнопку мыши над кнопкой Submit, генерируется событие, отличное от события, возникающего при нажатии кнопки мыши над кнопкой Reset формы.

В исходной событийной модели событие – это внутренняя абстракция для веб-броузера, и код JavaScript не может непосредственно манипулировать событием. Говоря о типе события в исходной событийной модели, мы на самом деле имеем в виду имя обработчика, вызываемого в ответ на событие. В этой модели код обработки событий задается с помощью атрибутов элементов HTML (и соответствующих свойств связанных с ними объектов JavaScript). Следовательно, если приложению требуется знать, что пользователь перемещает мышь над определенной гиперссылкой, то применяется атрибут `onmouseover` тега `<a>`, определяющего эту гиперссылку. А если приложению требуется знать, что пользователь щелкнул по кнопке Submit, используется атрибут `onclick` тега `<input>`, определяющего кнопку, или атрибут `onsubmit` элемента `<form>`, содержащего эту кнопку.

Имеется довольно много различных атрибутов обработчиков событий, которые можно использовать в исходной событийной модели. Они перечислены в табл. 19.1, в которой также указано, когда вызываются эти обработчики событий, и какие элементы HTML поддерживают атрибуты обработчиков.

В процессе развития программирования на клиентском JavaScript развивалась и поддерживаемая им событийная модель. В каждой новой версии броузера добавлялись новые атрибуты обработчиков событий. И наконец, спецификация HTML 4 закрепила стандартный набор атрибутов обработчиков событий для тегов HTML. В третьем столбце табл. 19.1 указано, какие элементы HTML поддерживают каждый из атрибутов обработчиков событий, а также какие версии броузеров поддерживают данный обработчик для данного тега и является ли обработчик события для данного тега стандартной частью HTML 4. В этом столбце «N» представляет собой сокращение для Nescape, а «IE» – для Internet Explorer. Любая версия броузера обратно совместима с предыдущими версиями, поэтому, например, «N3» обозначает Netscape 3 и все более поздние версии.

Внимательно изучив различные атрибуты обработчиков событий в табл. 19.1, можно выделить две обширных категории событий. Первая категория – это *события ввода* (*raw events или input events*). Эти события генерируются, когда пользователь перемещает мышь или нажимает кнопку мыши, или клавишу на клавиатуре. Эти низкоуровневые события просто описывают действия пользователя и не имеют другого смысла. Вторая категория событий – это *семантические события* (*semantic events*). Это высокоуровневые события, они имеют более сложный смысл и обычно происходят только в определенном

контексте: когда броузер завершил загрузку документа или, например, когда должна выполниться передача данных формы. Семантическое событие часто происходит как побочный эффект низкоуровневого события. Например, когда пользователь щелкает по кнопке мыши над кнопкой Submit, вызываются три обработчика событий ввода: onmousedown, onmouseup и onclick. И, как результат нажатия кнопки мыши, форма HTML, содержащая кнопку Submit, генерирует событие onsubmit.

И еще одно, последнее замечание по поводу табл. 19.1. Для обработчиков событий мыши в третьем столбце указано, что атрибут обработчика поддерживается (по крайней мере, в HTML 4) для «большинства элементов». Элементы HTML, не поддерживающие эти обработчики, обычно принадлежат к заголовку документа или не имеют собственного графического представления. Вот теги, не поддерживающие почти универсальные обработчики событий мыши: <applet>, <bdo>, <br>, <font>, <frame>, <frameset>, <head>, <html>, <iframe>, <isindex>, <meta> и <style>.

Таблица 19.1. Обработчики событий и поддерживающие их элементы HTML

Обработчик	Условие вызова	Поддерживается
onabort	Прерывание загрузки изображения	N3, IE4: <img>
onblur	Элемент теряет фокус ввода	HTML4, N2, IE3: <button>, <input>, <label>, <select>, <textarea> N3, IE4: <body>
onchange	Элемент <select> или другой элементе теряет фокус и его значение с момента получения фокуса изменилось	HTML4, N2, IE3: <input>, <select>, <textarea>
onclick	Была нажата и отпущена кнопка мыши; следует за событием mouseup. Возвращает false для отмены действия по умолчанию (то есть перехода по ссылке, очистки формы, передачи данных)	N2, IE3: <a>, <area>, <input> HTML4, N6, IE4: большинство элементов
ondblclick	Двойной щелчок	HTML4, N6, IE4: большинство элементов
onerror	Ошибка при загрузке изображения	N3, IE4: <img>
onfocus	Элемент получил фокус ввода	HTML4, N2, IE3: <button>, <input>, <label>, <select>, <textarea> N3, IE4: <body>
onkeydown	Клавиша нажата. Для отмены возвращает false	N4: <input>, <textarea> HTML4, N6, IE4: элементы формы и <body>
onkeypress	Клавиша нажата и отпущена. Для отмены возвращает false	N4: <input>, <textarea> HTML4, N6, IE4: элементы формы и <body>

Таблица 19.1 (продолжение)

Обработчик	Условие вызова	Поддерживается
onkeyup	Клавиша отпущена	N4: <input>, <textarea> HTML4, N6, IE4: элементы формы и <body>
onload	Загрузка документа завершена	HTML4, N2, IE3: <body>, <frameset> N3, IE4: <img> N6, IE4: <iframe>, <object>
onmousedown	Нажата кнопка мыши	N4: <a>, <area>, <img> HTML4, N6, IE4: большинство элементов
onmousemove	Перемещение курсора мыши	HTML4, N6, IE4: большинство элементов
onmouseout	Курсор мыши вышел за границы элемента	N3: <a>, <area> HTML4, N6, IE4: большинство элементов
onmouseover	Курсор мыши прошел над элементом. Для ссылок возвращает true, чтобы URL не появился в строке состояния	N2, IE3: <a>, <area> HTML4, N6, IE4: большинство элементов
onmouseup	Отпущена кнопка мыши	N4: <a>, <area>, <img> HTML4, N6, IE4: большинство элементов
onreset	Запрос на очистку полей формы. Для предотвращения очистки возвращает false	HTML4, N3, IE4: <form>
onresize	Изменение размеров окна	N4, IE4: <body>, <frameset>
Onselect	Выбор текста	HTML4, N6, IE3: <input>, <textarea>
onsubmit	Запрос на передачу данных формы. Чтобы предотвратить передачу, возвращает false	HTML4, N3, IE4: <form>
onunload	Документ или набор фреймов выгружен	HTML4, N2, IE3: <body>, <frameset>

19.1.2. Обработчики событий как атрибуты

Как мы видели в примерах из предыдущих глав, обработчики событий (в исходной событийной модели) задаются в виде строк кода JavaScript, присваиваемых как значения атрибутам HTML. Например, чтобы выполнить код JavaScript при щелчке по кнопке, укажите этот код в качестве значения атрибута onclick тега <input>:

```
<input type="button" value="Нажми меня" onclick="alert('спасибо');">
```

Значение атрибута обработчика события – это произвольная строка кода JavaScript. Если обработчик состоит из нескольких инструкций JavaScript, они *должны* отделяться друг от друга точками с запятой. Например:

```
<input type="button" value="Click Here"
       onclick="if (window.numclicks) numclicks++; else numclicks=1;
               this.value='Click # ' + numclicks;">
```

Если обработчик события требует нескольких инструкций, то как правило проще определить его в теле функции и затем задать атрибут HTML обработчика события для вызова этой функции. Например, проверить введенные пользователем в форму данные перед их отправкой можно при помощи атрибута `onsubmit` тега `<form>`. Проверка формы обычно требует как минимум нескольких строк кода, поэтому не надо помещать весь этот код в одно длинное значение атрибута, разумнее определить функцию проверки формы и просто задать атрибут `onsubmit` для вызова этой функции. Например, если для осуществления проверки определить функцию с именем `validateForm()`, то можно вызывать ее из обработчика события следующим образом:

```
<form action="processform.cgi" onsubmit="return validateForm();">
```

Помните, что HTML не чувствителен к регистру, поэтому в атрибутах обработчиков событий допускаются буквы любого регистра. Одно из распространенных соглашений состоит в употреблении символов различных регистров, при этом префикс «on» записывается в нижнем регистре: `onClick`, `onLoad`, `onMouseOut` и т. д. Однако в этой книге для обеспечения совместимости с XHTML, чувствительным к регистру я предпочел везде нижний регистр.

Код JavaScript в атрибуте обработчика события может содержать инструкцию `return`, а возвращаемое значение может иметь для броузера специальное значение. Вскоре мы это обсудим. Заметьте также, что код JavaScript обработчика события работает в области видимости (см. главу 4), отличной от глобального кода JavaScript. Это также более подробно обсуждается далее в этой главе.

19.1.3. Обработчики событий как свойства

Мы видели, что каждому элементу HTML в документе соответствует объект JavaScript в дереве документа, и что свойства этого объекта соответствуют атрибутам элемента HTML. В JavaScript 1.1 и более поздних версиях это также относится и к атрибутам обработчиков событий. Поэтому если в теге `<input>` имеется атрибут `onclick`, к указанному в нем обработчику событий можно обратиться с помощью свойства `onclick` объекта элемента формы. (JavaScript чувствителен к регистру, поэтому независимо от регистра символов, использовавшегося в имени атрибута HTML, свойство JavaScript должно быть записано целиком в нижнем регистре.)

Формально спецификация DOM не поддерживает исходную событийную модель, описанную здесь, и не определяет атрибуты JavaScript, соответствую-

щие атрибутам обработчиков событий, стандартизованных в HTML 4. Несмотря на отсутствие формальной стандартизации в DOM, эта событийная модель настолько распространена, что все поддерживающие JavaScript веб-броузеры допускают обращение к обработчикам событий как к свойствам JavaScript.

Так как значение атрибута HTML, определяющего обработчик событий, является строкой кода JavaScript, можно также предполагать, что соответствующее свойство JavaScript также представляет собой строку. Но это не так: при обращении через JavaScript свойства обработчиков событий представляют собой функции. Это проверяется с помощью простого примера:

```
<input type="button" value="Нажми меня" onclick="alert(typeof this.onclick);">
```

Если нажать кнопку, откроется диалоговое окно, содержащее слово «function», а не слово «string». (Обратите внимание, что в обработчиках событий ключевое слово `this` ссылается на объект, в котором произошло событие. Вскоре мы обсудим ключевое слово `this`.)

Для того чтобы назначить обработчик события элементу документа с помощью JavaScript, установите свойство-обработчик события равным нужной функции. Рассмотрим, например, следующую форму HTML:

```
<form name="f1">
<input name="b1" type="button" value="Нажми меня">
</form>
```

На кнопку в этой форме можно сослаться с помощью выражения `document.f1.b1`, значит, обработчик события можно установить с помощью следующей строки кода:

```
document.f1.b1.onclick=function() { alert('Спасибо!'); };
```

Кроме того, обработчик события может быть установлен вот так:

```
function plead() { window.status = "Пожалуйста, нажми меня!"; }
document.f1.b1.onmouseover = plead;
```

Обратите особое внимание на последнюю строку: здесь после имени функции нет скобок. Чтобы определить обработчик события, мы присваиваем свойству-обработчику события саму функцию, а не результат ее вызова. На этом часто спотыкаются начинающие программисты JavaScript.

В представлении обработчиков событий в виде свойств JavaScript есть два преимущества. Во-первых, это сокращает смешивание кода HTML и JavaScript, стимулируя модульность и позволяя получить более ясный и легко сопровождаемый код. Во-вторых, благодаря этому функции-обработчики событий являются динамическими. В отличие от атрибутов HTML, которые представляют собой статичную часть документа и могут устанавливаться только при создании документа, свойства JavaScript могут изменяться в любое время. В сложных интерактивных программах иногда полезно динами-

чески изменять обработчики событий, зарегистрированные для элементов HTML. Один небольшой недостаток определения обработчиков событий в JavaScript состоит в том то, что это отделяет обработчик от элемента, которому он принадлежит. Если пользователь взаимодействует с элементом документа до его полной загрузки (и до исполнения всех его сценариев), обработчики событий для элемента могут быть еще не определены.

В примере 19.1 показано, как назначить одну функцию обработчиком события для нескольких элементов документа. Данный пример представляет собой простую функцию, определяющую обработчик события `onclick` для каждой ссылки в документе. Обработчик события запрашивает подтверждение пользователя перед тем как разрешить переход по ссылке, по которой пользователь только что щелкнул. Если пользователь не дал подтверждения, функция-обработчик возвращает `false`, что не дает броузеру перейти по ссылке. Возвращаемые обработчиками событий значения будут обсуждаться в ближайших разделах.

Пример 19.1. Одна функция, много обработчиков событий

```
// Эта функция подходит для использования в качестве обработчика события
// onclick для элементов <a> и <area>. Она использует ключевое слово this
// для обращения к элементу документа и может возвращать false для отмены
// перехода по ссылке.
function confirmLink() {
  return confirm("Вы действительно хотите посетить " + this.href + "?");
}

// Эта функция выполняет цикл по всем гиперссылкам в документе и назначает
// каждой из них функцию confirmLink в качестве обработчика события.
// Не вызывайте ее до того, как документ будет проанализирован и все ссылки будут
// определены. Лучше всего вызвать ее из обработчика события onload тэга <body>.
function confirmAllLinks() {
  for(var i = 0; i < document.links.length; i++) {
    document.links[i].onclick = confirmLink;
  }
}
```

19.1.3.1. Явный вызов обработчиков событий

Значения свойств-обработчиков событий представляют собой функции, следовательно, их можно непосредственно вызывать при помощи JavaScript. Например, если для определения функции проверки формы мы задали атрибут `onsubmit` тега `<form>` и хотим проверить форму в какой-то момент до попытки передачи ее пользователем, мы можем обратиться к свойству `onsubmit` объекта Form для вызова функции-обработчика события. Код может выглядеть следующим образом:

```
document.myform.onsubmit();
```

Обратите, однако, внимание, что вызов обработчика события не является способом имитации действий, происходящих при реальном возникновении

этого события. Если, например, мы вызовем метод onclick объекта Link, это не заставит броузер перейти по ссылке и загрузить новый документ. Мы лишь выполним ту функцию, которую определили в качестве значения этого свойства. Чтобы заставить броузер загрузить новый документ, необходимо установить свойство location объекта Window, как мы видели в главе 13. То же самое справедливо для метода onsubmit объекта Form или метода onclick объекта Submit: вызов метода запускает функцию-обработчик события, но не приводит к передаче данных формы. (Чтобы на самом деле передать данные формы, мы вызываем метод submit() объекта Form.)

Одна из причин, по которой может быть нужен явный вызов функции-обработчика события – желание дополнить с помощью JavaScript обработчик события, который (возможно) уже определен кодом HTML. Предположим, что вы хотите предпринять специальные действия, когда пользователь щелкает по кнопке, но не хотите нарушать работу любого обработчика события onclick, который мог быть определен в самом HTML-документе. (Это один из недостатков кода в примере 19.1 – добавляя обработчик для каждой гиперссылки, он переписывает любые обработчики onclick, уже определенные для этих гиперссылок.) Этот результат достигается с помощью следующего кода:

```
var b = document.myform.mybutton;  // Это интересующая нас кнопка
var oldHandler = b.onclick;        // Сохраняем обработчик события HTML
function newHandler() { /* Здесь расположен мой код обработки события */ }
// Теперь назначаем новый обработчик события, вызывающий как новый, так и
// старый обработчик.
b.onclick = function() { oldHandler(); newHandler(); }
```

19.1.4. Значения, возвращаемые обработчиками событий

Во многих случаях обработчик события (заданный либо атрибутом HTML, либо свойством JavaScript) использует возвращаемое значение для указания поведения. Например, если с помощью обработчика события onsubmit объекта Form выполняется проверка формы и выясняется, что пользователь заполнил не все поля, то можно вернуть из обработчика false, чтобы предотвратить фактическую передачу данных формы. Гарантировать, что форма с пустым текстовым полем не будет передана, можно с помощью следующего кода:

```
<form action="search.cgi"
      onsubmit="if (this.elements[0].value.length == 0) return false;">
<input type="text">
</form>
```

Как правило, если в ответ на событие броузер выполняет некоторые действия по умолчанию, можно вернуть false, чтобы предотвратить выполнение этого действия. Вернуть false для отмены действия по умолчанию можно также из обработчиков событий onclick, onkeydown, onkeypress, onmousedown, onmouseup и onreset. Второй столбец табл. 19.1 содержит информацию о возвращаемых значениях этих обработчиков событий.

Из правила о возвращении false для отмены действия есть одно исключение: когда пользователь перемещает мышь над гиперссылкой (или изображением-картой), по умолчанию броузер отображает URL ссылки в строке состояния. Чтобы этого не случилось, необходимо вернуть true из обработчика onmouseover. Например, можно вывести сообщение, отличное от URL, с помощью следующего кода:

```
<a href="help.htm" onmouseover="window.status='Помогите!!'; return true;">Help</a>
```

Никакой особой причины для этого исключения нет: это так просто потому, что всегда было так.

Обратите внимание, что от обработчиков событий никогда не требуется обязательно явно возвращать значение. Если значение не возвратить, то выполняется действие по умолчанию.

19.1.5. Обработчики событий и ключевое слово this

Если обработчик события определен с помощью атрибута HTML или свойства JavaScript, ваши действия состоят в присваивании функции свойству элемента документа. Другими словами, вы определяете новый метод элемента документа. Ваш обработчик события вызывается как метод элемента, в котором произошло событие, поэтому ключевое слово this ссылается на этот целевой элемент. Это поведение полезно и не удивительно.

Однако убедитесь, что понимаете следствия этого поведения. Предположим, у вас есть объект o с методом mymethod. Обработчик события можно зарегистрировать следующим образом:

```
button.onclick= o.mymethod;
```

В результате выполнения этой инструкции button.onclick ссылается на ту же функцию, что и o.mymethod. Эта функция теперь является методом и для o и для button. Вызывая этот обработчик события, броузер вызывает функцию как метод объекта button, а не метод объекта o. Ключевое слово this ссылается на объект Button, а не на ваш объект o. Не делайте ошибку, думая, что можно обмануть броузер, вызывая обработчик события как метод какого-либо другого объекта. Для того чтобы сделать это, необходимо дать явное указание, например, так:

```
button.onclick = function() { o.mymethod(); }
```

19.1.6. Область видимости обработчиков событий

Как говорилось в главе 11, функции в JavaScript относятся к лексическому контексту. Это значит, что они работают в той области видимости, в которой они определены, а не в той, из которой вызываются. Если обработчик события определяется путем присваивания атрибуту HTML строки кода JavaScript, то при этом неявно определяется функция JavaScript (что можно было видеть при проверке типа соответствующего свойства-обработчика собы-

тия в JavaScript). Важно понимать, что область видимости для обработчика события, определенного подобным образом, не совпадает с областью видимости других глобальных функций JavaScript, определенных обычным образом. Это значит, что обработчики событий, определенные в виде атрибутов HTML, выполняются в контексте, отличном от контекста других функций.[1]

Вспомните, в главе 4 мы говорили, что область видимости функции определяется цепочкой областей видимости или списком объектов, которые по очереди просматриваются при поиске определения переменной. Когда переменная `x` разыскивается или разрешается в обычной функции, JavaScript сначала ищет локальную переменную или аргумент, проверяя объект вызова функции на наличие свойства с таким именем. Если такое свойство не найдено, JavaScript переходит к следующему объекту в цепочке областей видимости: глобальному объекту. Интерпретатор проверяет свойства глобального объекта, чтобы выяснить, является ли эта переменная глобальной.

Обработчики событий, определенные как атрибуты HTML, имеют более сложную цепочку областей видимости, чем описанная выше. Началом цепочки областей видимости является объект вызова. Здесь определены любые аргументы, переданные обработчику события (далее в этой главе мы увидим, что в некоторых развитых событийных моделях обработчикам событий передается аргумент), а также любые локальные переменные, определенные в теле обработчика. Следующим объектом в цепочке областей видимости является, однако, не глобальный объект, а объект, который вызвал обработчик события. Предположим, что объект Button в форме HTML определяется с помощью тега `<input>`, а затем посредством задания атрибута `onclick` определяется обработчик события. Если в коде обработчика есть переменная с именем `form`, то она разрешается в свойство `form` объекта Button. Это может быть удобным при создании обработчиков событий в виде атрибутов HTML.

Цепочка областей видимости обработчика события не заканчивается на объекте, определяющем обработчик события: она продолжается вверх по иерархии. Для описанного выше обработчика события `onclick` цепочка областей действия начинается с объекта вызова функции-обработчика. Затем, как мы видели, она продолжается объектом Button. После этого она продолжается по иерархии элементов HTML и включает, как минимум, элемент HTML `<form>`, содержащий кнопку, и объект Document, содержащий форму. Точный состав цепочки областей видимости никогда не был стандартизован и зависит от реализации. Netscape 6 и Mozilla включают все объекты-контейнеры (даже такие, как теги `<div>`), а IE 6 ограничивается более узким набором, включающим целевой элемент плюс содержащий его объект Form (если он существует) и объект Document. Независимо от броузера последним объектом в цепочке областей является объект Window, как и всегда в клиентском JavaScript.

---

[1] Это важно понимать, и хотя приведенное ниже обсуждение интересно, оно довольно сложное. При первом прочтении этой главы можно его пропустить и вернуться к нему позднее.

Наличие целевого объекта в цепочке областей действия может быть полезным. Но наличие расширенной цепочки областей действия, включающей другие элементы документа, может оказаться досадным неудобством. Заметьте, например, что и объект Window и объект Document определяют методы с именем `open()`. Если идентификатор `open` применяется без уточнения, то почти всегда имеет место обращение к методу `window.open()`. Однако в обработчике события, определенном как атрибут HTML, объект Document расположен в цепочке областей видимости до объекта Window, и отдельный идентификатор `open` будет ссылаться на метод `document.open()`. Аналогично, посмотрим, что будет происходить, если добавить свойство с именем `window` к объекту Form (или определить поле ввода с `name=»window»`). В этом случае, если определить внутри формы обработчик события, в котором есть выражение `window.open()`, идентификатор `window` разрешится как свойство объекта Form, а не как глобальный объект Window, и у обработчиков событий внутри формы не будет простого способа для обращения к глобальному объекту Window или вызова метода `window.open()`!

Мораль в том, что при определении обработчиков событий как атрибутов HTML необходима аккуратность. Подобным образом лучше создавать только очень простые обработчики. В идеале они должны просто вызывать глобальную функцию, определенную в другом месте, и, возможно, возвращать ее результат:

```
<script>
function validateForm() {
  /* Код проверки формы */
  return true;
}
</script>
<input type="submit" onclick="return validateForm();">
```

При исполнении простого обработчика событий, подобного приведенному выше, все еще действует цепочка областей видимости, отличающаяся от обычной, и можно нарушить функционирование обработчика, определив метод `validateForm()` в одном из элементов, содержащих данный. Но если предположить, что необходимая глобальная функция все-таки вызвана, она будет исполняться в глобальной области видимости. Несмотря на это, следует помнить, что при исполнении функций используется область видимости, в которой они определены, а не в область видимости, из которой они вызываются. Поэтому даже несмотря на то что метод `validateForm()` вызывается из области видимости, которая отличается от обычной, он будет выполняться в собственной глобальной области видимости.

Кроме того, поскольку нет стандарта на точный состав цепочки областей видимости для обработчика события, лучше всего предполагать, что она содержит только целевой элемент и глобальный объект Window. Например, ссылаться на целевой элемент посредством `this`, а если целевым является элемент `<input>`, ссылаться на содержащий его объект Form с помощью `form`. Но не полагайтесь на наличие в цепочке областей видимости объектов Form или

Document. Например, не используйте идентификатор `write` без спецификации для обращения к методу `write()` документа, а точно указывайте интерпретатору, что имеете в виду `document.write()`.

Не забывайте, что все это обсуждение области видимости обработчика события применимо только к обработчикам событий, определенным как атрибуты HTML. Если обработчик события задается путем присваивания функции соответствующему свойству-обработчику события, то никакой специальной цепочки областей видимости не возникает, и функция выполняется в той области видимости, в которой она была определена. Это почти всегда глобальная область видимости, если только это не вложенная функция, тогда цепочка областей действия снова становится интересной!

19.2. Развитые возможности обработки событий в DOM Level 2

Технологии обработки событий, рассмотренные в этой главе, являются частью DOM Level 0 – стандартного API, поддерживаемого любым броузером, который поддерживает JavaScript. Стандарт DOM Level 2 определяет развитый API обработки событий, значительно отличающийся (и значительно более мощный), чем API Level 0. Стандарт Level 2 не присоединяет существующий API к стандарту DOM, но и не отменяет API Level 0. Базовые задачи обработки событий по-прежнему можно решать средствами простого API.

Модуль Events DOM Level 2 поддерживается в Mozilla и в Netscape 6, но не поддерживается в Internet Explorer 6.

19.2.1. Распространение событий

В событийной модели Level 0 броузер передает события тем элементам документа, в которых они происходят. Если объект имеет соответствующий обработчик события, этот обработчик запускается. И больше ничего не происходит. В DOM Level 2 ситуация сложнее. В развитой событийной модели, когда событие происходит в узле документа (известном как *целевой узел* события), вызывается обработчик (или обработчики) событий целевого узла, но, кроме того, каждый из узлов-предков этого узла имеет одну или две возможности обработать данное событие. Распространение события осуществляется в три этапа. Сначала, на этапе *перехвата*, события распространяются от объекта Document вниз по дереву документа к целевому узлу. Если у какого-либо из предков целевого узла (но не у него самого) есть специально зарегистрированный перехватывающий обработчик события, эти обработчики запускаются на данном этапе распространения события. (Скоро мы узнаем, как регистрируются обычные и перехватывающие обработчики событий.)

Следующий этап распространения события происходит в самом целевом узле: запускаются любые соответствующие обработчики событий, зарегистри-

рованные непосредственно в целевом узле. Этот этап аналогичен обработке событий, предоставляемой моделью событий Level 0.

Третий этап распространения события – это этап «всплывания», на котором событие распространяется, или «всплывает», обратно, вверх по иерархии документа от целевого элемента к объекту Document. Если на этапе перехвата по дереву документа распространяются все события, то в этапе «всплывания» участвуют не все типы событий: например, событие `submit` не имеет смысла распространять вверх по документу за рамки элемента `<form>`, к которому оно относится. С другой стороны, универсальные события, такие как события `mousedown`, могут быть интересны любому элементу документа, поэтому они всплывают по иерархии документа, вызывая любые соответствующие обработчики событий во всех предках целевого элемента. Как правило, события ввода «всплывают», а высокоуровневые семантические события – нет. (Полный список всплывающих и невсплывающих событий приведен в табл. 19.3 далее в этой главе.)

Любой обработчик может остановить дальнейшее распространение события, вызвав метод `stopPropagation()` объекта Event, представляющего данное событие. Более подробную информацию об объекте Event и его методе `stopPropagation()` можно найти далее в этой главе.

Некоторые события приводят к выполнению веб-броузером связанных с ними действий по умолчанию. Например, когда в теге `<a>` происходит событие `click`, действием броузера по умолчанию является переход по гиперссылке. Такие действия по умолчанию выполняются только после завершения всех трех фаз распространения события, и любой обработчик, вызванный во время распространения события, имеет возможность отменить действие по умолчанию, вызвав метод `preventDefault()` объекта Event.

Такая схема распространения событий может показаться витиеватой, но она обеспечивает важные средства централизации кода обработки событий. DOM Level 2 предоставляет доступ ко всем элементам документа и допускает возникновение событий (таких как события `mouseover`) в любом из них. Это значит, что имеется намного больше мест, где могут быть зарегистрированы обработчики событий, чем в старой событийной модели Level 0. Предположим, что вы хотите вызывать обработчик события при перемещении курсора мыши над элементом `<p>` в вашем документе. Вместо регистрации обработчика события `onmouseover` для каждого тега `<p>` можно зарегистрировать один обработчик события в объекте Document и обрабатывать эти события либо на этапе перехвата, либо на этапе всплывания при распространении события.

И еще одна важная деталь, относящаяся к распространению событий. В модели Level 0 можно зарегистрировать только один обработчик для определенного типа события в определенном объекте. А в модели Level 2 можно зарегистрировать произвольное количество функций-обработчиков для определенного типа события в определенном объекте. Это также относится к предкам целевого узла события, чья функция или функции обработки вызывается во время этапов перехвата и всплывания распространения документа.

19.2.2. Регистрация обработчиков событий

В API Level 0 можно зарегистрировать обработчик события, установив значение атрибута в коде HTML или значение свойства объекта в коде JavaScript. В модели событий Level 2 обработчик события регистрируется для определенного элемента с помощью вызова метода `addEventListener()` этого объекта. (Стандарт DOM определяет для этого API термин «слушатель» (listener), но мы по-прежнему будем оперировать синонимом «обработчик» (handler).) Этот метод принимает три аргумента. Первый – имя типа события, для которого регистрируется обработчик. Тип события должен быть строкой, содержащей имя HTML-атрибута обработчика в нижнем регистре, без начальных букв «on». Другими словами, если в модели Level 0 применяется атрибут HTML `onmousedown` или свойство `onmousedown`, то в модели событий Level 2 необходимо использовать строку «mousedown».

Второй аргумент `addEventListener()` представляет собой функцию-обработчик (или слушатель), которая должна вызываться при возникновении указанного типа событий. Когда вызывается ваша функция, ей в качестве единственного аргумента передается объект Event. Этот объект содержит информацию о событии (например, какая кнопка мыши была нажата) и определяет методы, такие как `stopPropagation()`. Позднее мы узнаем об интерфейсе Event и его подынтерфейсах больше.

Последний аргумент метода `addEventListener()` – логическое значение. Если оно равно `true`, указанный обработчик события перехватывает события при их распространении на этапе перехвата. Если аргумент равен `false`, значит, это нормальный обработчик события и он вызывается, когда событие происходит непосредственно в данном элементе или в потомке элемента и затем всплывает обратно к данному элементу.

Например, вот так при помощи функции `addEventListener()` можно зарегистрировать обработчик события `submit` элемента `<form>`:

```
document.myform.addEventListener("submit",
                                 function(e) { validate(e.target); }
                                 false);
```

Можно перехватить все события `mousedown`, происходящие внутри элемента `<div>` с определенным именем, вызвав `addEventListener()` следующим образом:

```
var mydiv = document.getElementById("mydiv");
mydiv.addEventListener("mousedown", handleMouseDown, true);
```

В этих примерах предполагается, что функции `validate()` и `handleMouseDown()` определены в каком-либо другом месте вашего кода JavaScript.

Обработчики событий, зарегистрированные с помощью `addEventListener()`, выполняются в той области видимости, в которой они определены. Они не вызываются с расширенной цепочкой областей видимости, используемой для обработчиков событий, определенных в виде атрибутов HTML. (См. раздел 19.1.6 «Область видимости обработчиков событий».)

Обработчики событий в модели Level 2 регистрируются путем вызова метода, а не путем установки атрибута или свойства, поэтому можно зарегистрировать несколько обработчиков для данного типа события в данном объекте. Если вызвать `addEventListener()` несколько раз для регистрации нескольких функций-обработчиков для одного типа события в одном объекте, то при возникновении события этого типа в этом объекте (или при всплывании к данному объекту, или при перехвате данного события данным объектом) будут вызваны все зарегистрированные вами функции. Важно понимать, что стандарт DOM не гарантирует порядок, в котором вызываются функции-обработчики данного объекта, поэтому не следует рассчитывать, что они будут вызваны в том порядке, в каком вы их зарегистрировали. Заметьте также, что если несколько раз зарегистрировать в одном элементе одну и ту же функцию-обработчик, то все регистрации, кроме первой, игнорируются.

Зачем может потребоваться более одной функции-обработчика для одного события в одном объекте? Это может быть очень полезно для структуризации вашего программного кода. Предположим, что вы пишете универсальный модуль кода JavaScript, использующий события `mouseover` в изображениях для переключения картинок. Теперь предположим, что у вас есть еще один модуль, который хочет использовать те же события `mouseover` для вывода дополнительной информации об изображении (или о представляемой изображением ссылке) в строку состояния броузера. В случае применения API Level 0 придется объединить два модуля в один, чтобы они могли делить одно свойство `onmouseover` объекта Image. С другой стороны, в API Level 2 каждый модуль может зарегистрировать нужный ему обработчик события, не зная о другом модуле и не вмешиваясь в его работу.

Пару с методом `addEventListener()` образует метод `removeEventListener()`, требующий тех же трех аргументов, но не добавляющий, а удаляющий функцию обработки события из объекта. Часто бывает полезно временно зарегистрировать обработчик события, а потом удалить его. Например, при наступлении события `mousedown` можно зарегистрировать временные перехватывающие обработчики для событий `mousemove` и `mouseup`, чтобы видеть, перемещает ли пользователь курсор мыши. Затем, когда будет получено событие `mouseup`, необходимо отменить регистрацию этих обработчиков. При этом код удаления обработчика события может выглядеть следующим образом:

```
document.removeEventListener("mousemove", handleMouseMove, true);
document.removeEventListener("mouseup", handleMouseUp, true);
```

Методы `addEventListener()` и `removeEventListener()` определены интерфейсом EventTarget. В веб-броузерах, поддерживающих Event API из DOM Level 2, этот интерфейс реализован для всех узлов документа. Дополнительную информацию об этих методах регистрации и отмены регистрации можно найти в разделе по интерфейсу EventTarget в «Справочнике по W3C DOM».

И последнее замечание о регистрации обработчиков событий: в DOM Level 2 обработчики событий не ограничены элементами документа; можно также регистрировать обработчики для узлов Text. Однако на практике может ока-

заться проще зарегистрировать обработчики в элементах-контейнерах и допускать всплывание событий узла Text и их обработку на уровне контейнера.

19.2.3. Метод addEventListener и ключевое слово this

В исходной модели событий Level 0, когда функция регистрируется как обработчик события для элемента документа, она становится методом этого элемента (как это обсуждалось ранее в разделе «Область видимости обработчиков событий»). Когда вызывается обработчик события, он вызывается как метод элемента, и внутри функции ключевое слово `this` ссылается на элемент, в котором произошло событие.

В Mozilla и Netscape 6, когда функция-обработчик события регистрируется с помощью метода `addEventListener()`, она ведет себя так же: когда броузер вызывает функцию, она вызывается как метод элемента документа, для которого она была зарегистрирована. Однако заметьте, что это поведение зависит от броузера, и спецификация DOM не требует, чтобы это было так. Поэтому модели событий Level 2 не следует полагаться на значение ключевого слова `this` в собственных функциях-обработчиках событий. Вместо этого используйте свойство `currentTarget` объекта Event, который передается вашим функциям-обработчикам. Как мы увидим при рассмотрении объекта Event далее в этой главе, свойство `currentTarget` ссылается на объект, в котором был зарегистрирован обработчик события, но делает это переносимым образом.

19.2.4. Регистрация объектов в качестве обработчиков событий

Метод `addEventListener()` позволяет регистрировать функции-обработчики событий. Как обсуждалось в предыдущем разделе, будут ли вызываться эти функции как методы объектов, для которых они зарегистрированы, зависит от реализации. В объектно-ориентированном программировании можно определять обработчики событий как методы специального объекта и затем вызывать их как методы этого объекта. Для программистов Java стандарт DOM допускает именно это: в нем указано, что обработчики событий – это объекты, реализующие интерфейс EventListener и метод по имени `handleEvent()`. При регистрации обработчика события в Java методу `addEventListener()` передается объект, а не функция. Для простоты привязка DOM API в JavaScript не требует от нас реализации интерфейса EventListener, и позволяет вместо этого передавать методу `addEventListener()` прямые ссылки на функции.

Если в объектно-ориентированной программе на JavaScript объекты выступают в качестве обработчиков событий, то можно применять для их регистрации следующую функцию:

```
function registerObjectEventHandler(element, eventtype, listener, captures) {
    element.addEventListener(eventtype,
                             function(event) { listener.handleEvent(event); }
                             captures);
}
```

С помощью этой функции любой объект может быть зарегистрирован как слушатель событий, если в нем определен метод `handleEvent()`. Этот метод вызывается как метод объекта-слушателя, и ключевое слово `this` ссылается на этот объект, а не на элемент документа, сгенерировавший событие. Эта функция работает, так как использует вложенный функциональный литерал для перехвата и содержит в своей цепочке областей видимости объект-слушатель. (Те, кому это непонятно, могут вернуться к разделу 11.4 «Лексический контекст и вложенные функции».)

Хотя это не является частью спецификации DOM, Mozilla 0.9.1 и Netscape 6.1 (но не Netscape 6.0 и 6.01) допускают передачу объектов-слушателей событий, определяющих метод `handleEvent()`, непосредственно в метод `addEventListener()` вместо ссылки на функцию. Для этих броузеров специальная функция регистрации, подобная той, которую мы определяли выше, не нужна.

19.2.5. Модули и типы событий

Как уже говорилось, DOM Level 2 имеет модульную структуру, поэтому реализация может поддерживать одни его части, но исключать поддержку других. Events API – один из таких модулей. Проверить, поддерживает ли броузер этот модуль, можно посредством следующего кода:

```
document.implementation.hasFeature("Events", "2.0")
```

Однако модуль Events содержит только API для базовой инфрастуктуры обработки событий. Поддержка определенных типов событий делегируется субмодулям. Каждый субмодуль предоставляет поддержку определенной категории связанных типов событий и определяет тип Event, передаваемый обработчикам событий каждого из этих типов. Например, субмодуль по имени MouseEvents предоставляет поддержку `mousedown`, `mouseup`, `click` и родственных типов событий. Он также определяет интерфейс MouseEvent. Объект, реализующий этот интерфейс, передается функции-обработчику любого типа события, поддерживаемого данным модулем.

В табл. 19.2 перечислены все модули событий, определяемые ими интерфейсы и типы событий, которые они поддерживают. Заметьте, что DOM Level 2 не стандартизует ни один тип клавиатурного события, поэтому в данном списке нет модуля клавиатурных событий. Поддержка этого типа событий ожидается в стандарте DOM Level 3.

Таблица 19.2. Модули, интерфейсы и типы событий

Имя модуля	Интерфейс Event	Типы событий
HTMLEvents	Event	abort, blur, change, error, focus, load, reset, resize, scroll, select, submit, unload
MouseEvents	MouseEvent	click, mousedown, mousemove, mouseout, mouseover, mouseup
UIEvents	UIEvent	DOMActivate, DOMFocusIn, DOMFocusOut

Таблица 19.2 (продолжение)

Имя модуля	Интерфейс Event	Типы событий
MutationEvents	MutationEvent	DOMAttrModified, DOMCharacterDataModified, DOMNodeInserted, DOMNodeInsertedIntoDocument, DOMNodeRemoved, DOMNodeRemovedFromDocument, DOMSubtreeModified

Как видно из табл. 19.2, модули HTMLEvents и MouseEvents определяют типы событий, схожие с модулем событий Level 0. Модуль UIEvents определяет типы событий, схожие с событиями `focus`, `blur` и `click`, поддерживаемыми элементами форм HTML, но обобщенные таким образом, что они могут генерироваться любым элементом документа, который может получать фокус или активизироваться как-то иначе. Модуль MutationEvents определяет события, генерируемые при каком-либо изменении документа. Это специальные типы событий, и они используются достаточно редко.

Мы уже говорили, что когда происходит событие, его обработчику передается объект, реализующий интерфейс Event, который связан с данным типом события. Свойства этого объекта предоставляют информацию о событии, которая может быть полезна обработчику. В табл. 19.3 снова перечислены стандартные события, но на этот раз организованные по типам, а не по модулям событий. Для каждого типа событий в этой таблице указан вид объекта события, передаваемого его обработчику, а также говорится, всплывает ли событие этого типа по иерархии документа в процессе распространения события (столбец «B») и есть ли для данного события действие по умолчанию, которое может быть отменено методом `preventDefault()` (столбец «C»). Для событий модуля HTMLEvents в пятом столбце таблицы указано, какие элементы HTML могут генерировать данное событие. Для всех остальных типов событий в пятом столбце указано, какие свойства объекта события содержат существенные подробности о событии (эти свойства описаны в следующем разделе). Обратите внимание, что свойства, перечисленные в этом столбце, не включают свойств, определяемых базовым интерфейсом Event и содержащих осмысленные значения для всех типов событий.

Полезно сравнить табл. 19.3 с табл. 19.1, в которой перечислены обработчики событий Level 0, определяемые в HTML 4. Типы событий, поддерживаемые двумя этими моделями, в значительной степени совпадают (исключая модули UIEvents и MutationEvents). Стандарт DOM Level 2 добавляет поддержку типов событий `abort`, `error`, `resize` и `scroll`, которые не были стандартизованы в HTML 4, и не поддерживает тип события `dblclick`, являющегося частью стандарта HTML 4. (Вместо этого, как мы скоро увидим, свойство `detail` объекта, передаваемого обработчику события `click`, указывает количество последовательных нажатий клавиши мыши.)

Таблица 19.3. Типы событий

Тип события	Интерфейс	B	C	Поддерживается/ детализирующие свойства
abort	Event	да	нет	`<img>`, `<object>`
blur	Event	нет	нет	`<a>`, `<area>`, `<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`
change	Event	да	нет	`<input>`, `<select>`, `<textarea>`
click	MouseEvent	да	да	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `button`, `detail`
error	Event	да	нет	`<body>`, `<frameset>`, `<img>`, `<object>`
focus	Event	нет	нет	`<a>`, `<area>`, `<button>`, `<input>`, `<label>`, `<select>`, `<textarea>`
load	Event	нет	нет	`<body>`, `<frameset>`, `<iframe>`, `<img>`, `<object>`
mousedown	MouseEvent	да	да	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `button`, `detail`
mousemove	MouseEvent	да	нет	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`
mouseout	MouseEvent	да	да	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `relatedTarget`
mouseover	MouseEvent	да	да	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `relatedTarget`
mouseup	MouseEvent	да	да	`screenX`, `screenY`, `clientX`, `clientY`, `altKey`, `ctrlKey`, `shiftKey`, `metaKey`, `button`, `detail`

Таблица 19.3 (продолжение)

Тип события	Интерфейс	B	C	Поддерживается/ детализирующие свойства
reset	Event	да	нет	`<form>`
resize	Event	да	нет	`<body>`, `<frameset>`, `<iframe>`
scroll	Event	да	нет	`<body>`
select	Event	да	нет	`<input>`, `<textarea>`
submit	Event	да	да	`<form>`
unload	Event	нет	нет	`<body>`, `<frameset>`
DOMActivate	UIEvent	да	да	`detail`
DOMAttrModified	MutationEvent	да	нет	`attrName`, `attrChange`, `prevValue`, `newValue`, `relatedNode`
DOMCharacterDataModified	MutationEvent	да	нет	`prevValue`, `newValue`
DOMFocusIn	UIEvent	да	нет	отсутствуют
DOMFocusOut	UIEvent	да	нет	отсутствуют
DOMNodeInserted	MutationEvent	да	нет	`relatedNode`
DOMNodeInsertedIntoDocument	MutationEvent	нет	нет	отсутствуют
DOMNodeRemoved	MutationEvents	да	нет	`relatedNode`
DOMNodeRemovedFromDocument	MutationEvent	нет	нет	отсутствуют
DOMSubtreeModified	MutationEvent	да	нет	отсутствуют

19.2.6. Интерфейсы и детализирующие свойства событий

Когда происходит событие, DOM Level 2 API предоставляет дополнительные подробности (например, где и когда оно произошло) о нем в виде свойств объекта, передаваемого обработчику события. С каждым модулем событий связан интерфейс событий, в котором содержатся детали, относящиеся к этому типу событий. В табл. 19.2 перечислены четыре различных модуля и четыре различных интерфейса событий.

Эти четыре интерфейса фактически связаны друг с другом и образуют иерархию. Интерфейс Event является вершиной иерархии; все объекты событий реализуют этот базовый интерфейс. UIEvent – это подынтерфейс Event: любой объект события, реализующий UIEvent, также реализует все методы и свойства Event. Интерфейс MouseEvent является подынтерфейсом UIEvent. Это значит, например, что объект события, переданный обработчику события `click`, реализует все методы и свойства, определенные в каждом из интерфейсов MouseEvent, UIEvent и Event. И наконец, интерфейс MutationEvent является подынтерфейсом Event.

В следующих разделах представлен каждый из интерфейсов событий и выделены их наиболее важные свойства и методы. Полные описания всех интерфейсов можно найти в «Справочнике по W3C DOM» в этой книге (часть V).

19.2.6.1. Event

Типы событий, определенные в модуле HTMLEvents, используют интерфейс Event. Все остальные типы событий используют подынтерфейсы этого интерфейса, то есть интерфейс Event реализуется всеми объектами событий и предоставляет детальную информацию, применимую ко всем типам событий. Интерфейс Event определяет следующие свойства (заметьте, что эти свойства и свойства всех подынтерфейсов Event доступны только для чтения):

`type`

Тип произошедшего события. Значением этого свойства является имя типа события, и это та же строка, которая была использована при регистрации обработчика события (например, «click» или «mouseover»).

`target`

Узел документа, в котором произошло событие; может не совпадать с `currentTarget`.

`currentTarget`

Узел, в котором в данный момент обрабатывается событие (то есть узел, чей обработчик события работает в данный момент). Если событие обрабатывается на этапах перехвата или всплывания события, значение этого свойства отличается от значения свойства `target`. Как говорилось ранее, необходимо использовать это свойство в собственных функциях обработки событий вместо ключевого слова `this`.

`eventPhase`

Число, указывающее, какой этап распространения события выполняется в данный момент. Значением является одна из констант `Event.CAPTURING_PHASE` (этап перехвата), `Event.AT_TARGET` (обработка в целевом элементе) или `Event.BUBBLING_PHASE` (этап всплывания).

`timeStamp`

Объект Date, указывающий, когда произошло событие.

`bubbles`

Логическое значение, указывающее, всплывает ли это событие (и события этого типа) вверх по дереву документов.

`cancelable`

Логическое значение, указывающее, связано ли с этим событием действие по умолчанию, которое может быть отменено с помощью метода `preventDefault()`.

В дополнение к этим семи свойствам в интерфейсе Event определены два метода. Они также реализуются всеми объектами событий: `stopPropagation()` и

preventDefault(). Любой обработчик события может вызвать метод stopPropagation() для предотвращения распространения события за пределы узла, в котором оно обрабатывается в данный момент. Любой обработчик события может вызвать метод preventDefault(), чтобы предотвратить выполнение броузером действия по умолчанию, связанного с событием. Вызов preventDefault() в API DOM Level 2 эквивалентен возвращению обработчиком false в модели событий Level 0.

19.2.6.2. UIEvent

Интерфейс UIEvent является подынтерфейсом Event. Он определяет тип объекта события, передаваемого событиям типа DOMFocusIn, DOMFocusOut и DOMActivate. Эти типы событий используются достаточно редко и, что более важно, интерфейс UIEvent является родительским интерфейсом для MouseEvent. UIEvent определяет два свойства в дополнение к свойствам, определяемым интерфейсом Event.

view

Объект Window (в терминологии DOM – «вид»), внутри которого произошло событие.

detail

Число, которое может предоставить дополнительную информацию о событии. Для событий click, mousedown и mouseup это поле содержит количество нажатий: 1 для одинарного нажатия, 2 – для двойного и 3 – для тройного. (Обратите внимание, что каждое нажатие кнопки мыши генерирует событие, но если несколько нажатий следуют достаточно быстро, на это указывает значение detail. То есть событию мыши с detail, равным 2, всегда предшествует событие мыши с detail, равным 1.) Для событий DOMActivate это поле равно 1 в случае нормальной активации и 2 – в случае гиперактивации, например двойного щелчка или нажатия комбинации клавиш Shift-Enter.

19.2.6.3. MouseEvent

Интерфейс MouseEvent наследует свойства и методы Event и UIEvent и определяет следующие дополнительные свойства:

button

Число, указывающее, какая кнопка мыши изменила свое состояние во время события mousedown, mouseup или click. Значение 0 обозначает левую кнопку, 1 – среднюю кнопку, а 2 – правую кнопку. Это свойство применяется, только когда кнопка изменяет состояние, и не применяется, например, для получения информации о том, нажата ли кнопка во время события mousemove. Заметьте также, что Netscape 6 ведет себя неправильно и использует вместо значений 0, 1 и 2 значения 1, 2 и 3. В Netscape 6.1 это исправлено.

altKey, ctrlKey, metaKey, shiftKey

Эти четыре логических значения указывают, были ли нажаты клавиши Alt, Ctrl, Meta или Shift, когда произошло событие мыши. В отличие от свойства button эти клавиатурные свойства действительны для любого типа события мыши.

clientX, clientY

Эти два свойства указывают координаты X и Y курсора мыши относительно клиентской области или окна броузера. Обратите внимание, что эти координаты не учитывают прокрутку документа: если событие происходит в верхнем крае окна, clientY равно 0, независимо от того, насколько далеко был прокручен документ. К сожалению, DOM Level 2 не предоставляет стандартного способа трансляции этих оконных координат в координаты документа. В Netscape 6 можно сложить значения window.pageXOffset и window.pageYOffset, а в Internet Explorer – значения document.body.scrollLeft и document.body.scrollTop.

screenX, screenY

Эти два свойства задают координаты X и Y курсора мыши относительно верхнего левого края дисплея. Эти значения полезны, если вы планируете открыть новое окно в месте возникновения события мыши или рядом с ним.

relatedTarget

Это свойство ссылается на узел, который связан с целевым узлом события. Для событий mouseover это узел, который курсор мыши оставил, когда перешел к целевому узлу. Для событий mouseout это узел, куда переместился курсор мыши, оставив целевой узел. Для других типов событий это свойство не используется.

19.2.6.4 MutationEvent

Интерфейс MutationEvent – это подынтерфейс Event, используемый для предоставления подробностей о типах событий, определяемых модулем MutationEvents. Эти типы событий редко используются в программировании DHTML, поэтому данный интерфейс подробно здесь не рассматривается (см. «Справочник по W3C DOM»).

19.2.7. Пример: буксировка элементов документа

Итак, мы обсудили распространение событий, регистрацию обработчиков и различных интерфейсов объектов событий для событийной модели DOM Level 2, и можем, наконец, посмотреть, как это все работает. В примере 19.2 показана функция JavaScript beginDrag(), которая, будучи вызвана из обработчика события mousedown, позволяет пользователю перетащить элемент документа.

Функция beginDrag() принимает два аргумента. Первый – это перетаскиваемый элемент. Это может быть элемент, в котором произошло событие mouse-

down, или элемент, содержащий его (например, можно разрешить пользователю, перетаскивая заголовок окна, переместить все окно). Однако в обоих случаях он должен ссылаться на элемент документа, абсолютно позиционируемый с помощью атрибута CSS position, а значения атрибутов CSS left и top должны быть явно установлены в пиксельные значения в атрибуте style. Второй аргумент – это объект события, связанный с вызываемым событием mousedown.

Функция beginDrag() записывает позицию, в которой произошло событие mousedown, и затем регистрирует обработчики для событий mousemove и mouseup, которые последуют за событием mousedown. Обработчик события mousemove отвечает за перемещение элемента документа, а обработчик события mouseup – за отмену регистрации себя и обработчика mousemove. Важно отметить, что обработчики mousemove и mouseup зарегистрированы как перехватывающие обработчики, так как пользователь может двигать мышь быстрее, чем элемент документа сможет за ней следовать, и некоторые из этих событий могут происходить вне исходного элемента документа. Кроме того, обратите внимание, что функции moveHandler() и upHandler(), регистрируемые для обработки этих событий, определены как вложенные внутри beginDrag(). Они определяются в этой вложенной области видимости и поэтому могут использовать аргументы и локальные переменные beginDrag(), что значительно упрощает их реализацию.

Пример 19.2. Буксировка с использованием событийной модели DOM Level 2

```
/**
 * Drag.js:
 * Эта функция предназначена для вызова из обработчика события mousedown.
 * Она регистрирует временные перехватывающие обработчики последующих событий
 * mousemove и mouseup и использует эти обработчики для буксировки
 * указанного элемента документа. Первый аргумент должен быть абсолютно
 * позиционируемым элементом документа. Это может быть либо элемент,
 * получивший событие mousedown, либо какой-либо содержащий его элемент.
 * Второй аргумент должен быть объектом события mousedown.
 **/
function beginDrag(elementToDrag, event) {
    // Выясняем, где сейчас находится элемент.
    // Элемент должен иметь свойства CSS left и top в атрибуте style
    // Также мы предполагаем, что они установлены с использованием пикселей
    // в качестве единиц измерения.
    var x = parseInt(elementToDrag.style.left);
    var y = parseInt(elementToDrag.style.top);

    // Вычисляем расстояние между этой точкой и местом нажатия кнопки мыши.
    // Эти значения потребуются вложенной функции moveHandler, определенной
    // ниже.
    var deltaX = event.clientX - x;
    var deltaY = event.clientY - y;

    // Регистрируем обработчики, которые будут реагировать на события
    // mousemove и mouseup, следующие за событием mousedown. Обратите внимание,
```

```
    // что они регистрируются как перехватывающие обработчики в объекте Document.
    // Эти обработчики остаются активными, пока нажата кнопка мыши, и удаляются,
    // когда кнопка мыши отпускается.
    Document.addEventListener("mousemove", moveHandler, true);
    Document.addEventListener("mouseup", upHandler, true);

    // Мы обработали это событие. Не разрешаем кому-либо еще его увидеть.
    event.stopPropagation();
    event.preventDefault();

    /**
     * Это обработчик, который перехватывает события mousemove во время
     * перетаскивания элемента. Он отвечает за перемещение элемента.
     **/
    function moveHandler(event) {
        // Перемещаем элемент к текущей позиции курсора мыши, по необходимости
        // подстраиваем его позицию на смещение начального нажатия.
        elementToDrag.style.left = (event.clientX - deltaX) + "px";
        elementToDrag.style.top = (event.clientY - deltaY) + "px";

        // И не даем кому-либо еще увидеть это событие.
        event.stopPropagation();
    }

    /**
     * Это обработчик, перехватывающий финальное событие mouseup, которое
     * возникает в конце перетаскивания.
     **/
    function upHandler(event) {
        // Отменяем регистрацию перехватывающих обработчиков событий
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
        // И не даем событию распространяться дальше.
        event.stopPropagation();
    }
}
```

Можно поместить функцию `beginDrag()` **в HTML-файл, подобный показанному ниже (который представляет собой упрощенную версию примера 19.2 с добавлением буксировки):**

```
<script src="Drag.js"></script> <!-Подключаем сценарий Drag.js -->
<!-- Определяем буксируемый элемент -->
<div style="position:absolute; left:100px; top:100px;
            background-color: white; border: solid black;">
<!-- Определяем обработчик, выполняющий буксировку. -->
Обратите внимание на атрибут onmousedown. -->
<div style="background-color: gray; border-bottom: dotted black;
            padding: 3px; font-family: sans-serif; font-weight: bold;"
     onmousedown="beginDrag(this.parentNode, event);">
Перетащи меня <!-- Содержимое "заголовка" -->
```

```
</div>
<!-- Содержимое буксируемого элемента -->
<p>Это проверка. Проверяем, проверяем, проверяем.<p>Это проверка.<p>Проверка.
</div>
```

Ключевым здесь является атрибут onmousedown внутреннего элемента . Несмотря на то что beginDrag() использует событийную модель DOM Level 2, мы регистрируем эту функцию, используя для удобства модель Level 0. В следующем разделе вы узнаете, что событийные модели можно смешивать, и что когда обработчик события указан как атрибут HTML, доступ к объекту события можно получить с помощью ключевого слова event. (Это соглашение является частью стандарта DOM, но оно принято в событийных моделях Netscape 4 и IE, описанных далее.)

Вот еще один простой пример применения beginDrag(); он определяет изображение, которое пользователь может перетащить, только если нажата клавиша Shift:

```
<script src="Drag.js"></script>
<img src="plus.gif" width="20" height="20"
style="position:absolute; left:0px; top:0px;"
onmousedown="if (event.shiftKey) beginDrag(this, event);">
```

Обратите внимание на разницу между атрибутом onmousedown здесь и в предыдущем примере.

19.2.8. Смешивание событийных моделей

До этого момента мы обсуждали традиционную событийную модель Level 0 и новую стандартную модель DOM Level 2. Для обеспечения обратной совместимости броузеры, поддерживающие модель Level 2, продолжают поддерживать событийную модель Level 0. Это значит, что можно смешивать событийные модели внутри документа, как мы сделали во фрагментах HTML, использованных для демонстрации сценария буксировки элементов в предыдущем разделе.

Важно понимать, что веб-броузеры, поддерживающие событийную модель Level 2, всегда передают объект события обработчикам событий – даже обработчикам событий, зарегистрированным в соответствии с моделью Level 0 путем установки атрибута HTML или свойства JavaScript. Когда обработчик события определяется как атрибут HTML, он неявно преобразуется в функцию, имеющую аргумент с именем event. Это значит, что такой обработчик события может использовать идентификатор event для ссылки на объект события.

В стандарте DOM никогда не была формализована событийная модель Level 0. Он не требует наличия таких свойств, как onclick, для элементов HTML, поддерживающих атрибут onclick. Однако стандарт понимает, что событийная модель Level 0 будет использоваться и далее, и указывает, что реализации, поддерживающие Level 0, рассматривают обработчики, зарегистрированные в этой модели так же, как если бы они были зарегистрированы с по-

мощью addEventListener(). Другими словами, если вы присваиваете функцию f свойству onclick элемента документа e (или устанавливаете соответствующий атрибут HTML onclick), это эквивалентно регистрации этой функции инструкцией

```
e.addEventListener("click", f, false);
```

Когда вызывается функция f(), ей в качестве аргумента передается объект события, даже если она зарегистрирована с помощью модели Level 0. Кроме того, если вы изменяете значение свойства onclick с функции f на функцию g, это эквивалентно следующему коду:

```
e.removeEventListener("click", f, false);
e.addEventListener("click", g, false);
```

Однако заметим, что спецификация Level 2 не говорит, может ли обработчик события, зарегистрированный путем присваивания свойству onclick, быть удален вызовом метода removeEventListener(). На момент написания этой книги реализация Mozilla/Netscape этого не допускает.

19.2.9. Синтезирование событий

Стандарт DOM Level 2 включает API для создания и диспетчеризации синтетических событий. Этот API позволяет генерировать события программным путем вместо генерации их в результате действий пользователя. Хотя эта возможность требуется не часто, она может быть полезна, например, для проведения тестов, воздействующих на приложение DHTML известной последовательностью событий и проверяющих, всегда ли одинаковый получается результат. Еще эта возможность может применяться для реализации средств воспроизведения макросов, автоматизирующих часто выполняемые действия в пользовательском интерфейсе. С другой стороны, API синтетических событий не подходит для создания самовыполняющихся демонстрационных программ: можно создать синтетическое событие mousemove и доставить его до обработчиков событий в вашем приложении, но это не приводит к фактическому перемещению курсора мыши по экрану!

К сожалению, на момент написания этой книги API синтетических событий не поддерживается ни в Netscape 6, ни в текущей версии Mozilla.

Для генерации синтетического события надо сделать три шага:

- Создание соответствующего объекта события.
- Инициализация полей объекта события.
- Передача объекта события необходимому элементу документа.

Для создания объекта события применяется метод createEvent() объекта Document. Этот метод принимает один аргумент, представляющий собой имя модуля события, для которого должен быть создать объект события. Например, чтобы создать объект, подходящий для использования в событии click, вызовите метод следующим образом:

```
document.createEvent("HTMLEvents");
```

Для создания объекта события, подходящего для применения с любым типом события мыши, вызовите его вот так:

```
document.createEvent("MouseEvents");
```

Обратите внимание, что аргумент `createEvent()` указывается во множественном числе. Это нелогично, но эта строка совпадает с той, которую вы бы передали методу `hasFeature()`, чтобы проверить, поддерживает ли броузер этот модуль событий.

Следующий после создания объекта события шаг – инициализация его свойств. Свойства объекта события, однако, всегда доступны только для чтения, поэтому нельзя присваивать значения непосредственно им. Вместо этого для выполнения инициализации необходимо вызвать метод. Хотя в приведенных ранее описаниях Event, MouseEvent и других объектов событий упоминались только свойства, каждый объект также определяет один метод для инициализации свойств события. Этот метод инициализации имеет имя, зависящее от типа инициализируемого объекта события, и ему передается столько аргументов, сколько свойств должно быть установлено. Обратите внимание, что можно вызвать метод инициализации события только перед диспетчеризацией синтетического события: нельзя с помощью этих методов изменять свойства объекта события, переданного обработчику.

Посмотрим на пару примеров. Как вы знаете, события `click` являются частью модуля HTMLEvents и используют объекты типа Event. Эти объекты инициализируются с помощью метода `initEvent()`:

```
e.initEvent("click", "true", "true");
```

События `mousedown` являются частью модуля MouseEvents и используют объекты типа MouseEvent. Эти объекты инициализируются с помощью метода `initMouseEvent()`, требующего значительно большего количества аргументов:

```
e.initMouseEvent("mousedown", true, false,  // Свойства Event
                 window, 1,                 // Свойства UIEvent
                 0, 0, 0, 0,                // Свойства MouseEvent
                 false, false, false, false,
                 0, null);
```

Обратите внимание, что методу `createEvent()` передается только имя модуля событий. Имя фактического типа события передается методу инициализации события. Стандарт DOM не требует, чтобы это было одно из предопределенных имен. Можно создавать события с любым выбранным именем типа, если только это имя не начинается с цифры или префикса «DOM» (в верхнем, нижнем или смешанном регистре). Инициализируя синтетическое событие со специальным именем типа, необходимо также регистрировать обработчики событий с этим именем типа события.

После создания и инициализации объекта события можно выполнить его диспетчеризацию, передав его методу `dispatchEvent()` соответствующего элемента документа. Метод `dispatchEvent()` определен в интерфейсе EventTarget, поэтому он доступен как метод любого узла документа, поддерживающего методы `addEventListener()` и `removeEventListener()`. Элемент, в который мы отправляем событие, становится целевым элементом события, и объект проходит обычную последовательность распространения события. На каждой стадии распространения события созданный объект события передается любым обработчикам, которые были зарегистрированы для типа события, которое было указано при его инициализации. И наконец, когда распространение события завершается, вызов `dispatchEvent()` также завершается. Возвращаемое им значение равно `false`, если какой-либо из обработчиков вызвал метод `preventDefault()` вашего объекта события, и `true` в противном случае.

19.3. Событийная модель Internet Explorer

Событийная модель, поддерживаемая Internet Explorer 4, 5, 5.5 и 6, является переходной, находящейся посередине между исходной моделью Level 0 и стандартной моделью DOM Level 2. Событийная модель IE включает объект Event, предоставляющий детали о произошедшем событии. Однако вместо передачи функциям обработки событий, объект Event сделан доступным в виде свойства объекта Window. Модель Internet Explorer поддерживает всплывание при распространении события, но не поддерживает перехват, как модель DOM. В броузере IE 4 обработчики событий регистрируются так же, как в исходной модели Level 0. Однако в IE 5 и более поздних версиях с помощью специальных (и нестандартных) функций регистрации могут быть зарегистрированы множественные обработчики.

В следующих разделах эта событийная модель представлена более подробно в сравнении с исходной событийной моделью Level 0 и стандартной событийной моделью Level 2. Следовательно, прежде чем читать описание модели IE, вы должны убедиться, что понимаете эти две модели.

19.3.1. Объект Event в IE

Как и стандартная событийная модель DOM Level 2, событийная модель IE предоставляет подробную информацию о каждом произошедшем событии в виде свойств объекта Event. Объекты Event, определенные в стандартной событийной модели, на самом деле спроектированы на основе объекта Event из IE, поэтому можно заметить много общего между свойствами объекта Event в IE и свойствами объектов Event, UIEvent и MouseEvent в DOM.

Вот наиболее важные свойства объекта Event в IE:

`type`

Строка, указывающая тип произошедшего события. Значение этого свойства совпадает с именем обработчика события без ведущих символов «on»

(например, «click» или «mouseover»). Совместимо со свойством `type` в событийной модели DOM.

`srcElement`

Элемент документа, в котором произошло событие. Сравнимо со свойством `target` объекта Event в DOM.

`button`

Целое число, обозначающее нажатую кнопку мыши. Значение 1 обозначает левую кнопку, 2 – правую кнопку, а 4 – среднюю кнопку мыши. Если нажато несколько кнопок, эти значения складываются вместе – например, левой и правой кнопкам, нажатым вместе, будет соответствовать значение 3. Сравните это со свойством `button` объекта MouseEvent DOM Level 2, но обратите внимание, что хотя имена свойств совпадают, интерпретация их значений различается.

`clientX`, `clientY`

Эти свойства совместимы со свойствами MouseEvent DOM Level 2 с теми же именами и указывают координаты курсора мыши в момент события, относительно левого верхнего угла окна. Для документов, имеющих больший размер, чем окно, эти координаты не совпадают с позицией в документе, и может потребоваться прибавить к ним, соответственно, `document.body.scrollLeft` и `document.body.scrollTop` для учета прокрутки.

`offsetX`, `offsetY`

Эти свойства указывают положение курсора мыши относительно исходного элемента. Они позволяют определить, например, на каком пикселе объекта Image произведено нажатие кнопки. Эти свойства не имеют эквивалента в событийной модели DOM Level 2.

`altKey`, `ctrlKey`, `shiftKey`

Эти логические свойства указывают, были ли нажаты клавиши Alt, Ctrl и Shift при возникновении события. Эти свойства с совместимы с одноименными свойствами объекта MouseEvent DOM Level 2. Заметьте, однако, что у объекта Event в IE нет свойства `metaKey`.

`keyCode`

Целочисленное свойство. Задает код клавиши для событий `keydown` и `keyup` и код символа Unicode для события `keypress`. Коды символов преобразуются в строки методом `String.fromCharCode()`. Модель событий DOM Level 2 не стандартизует клавиатурные события (но группа создания DOM Level 3 работает над этим) и не имеет эквивалента для этих свойств.

`fromElement`, `toElement`

Свойство `fromElement` указывает для события `mouseover` элемент документа, в котором находился курсор мыши. Свойство `toElement` указывает для события `mouseout` элемент документа, куда переместился курсор мыши. Сравнимы со свойством `relatedTarget` объекта MouseEvent DOM Level 2.

`cancelBubble`

Логическое свойство. Будучи установленным в `true`, предотвращает дальнейшее всплывание события вверх по иерархии включения элементов. Сравнимо с методом `stopPropagation()` объекта Event DOM Level 2.

`returnValue`

Логическое свойство, которое может быть установлено в `false` для предотвращения выполнения пользователем действия по умолчанию, связанного с событием. Это альтернатива традиционному приему возвращения `false` из обработчика события. Сравнимо с методом `preventDefault()` объекта Event DOM Level 2.

Полное описание объекта Event из IE приведено в «Справочнике по клиентскому JavaScript» в этой книге.

19.3.2. Объект Event в IE как глобальная переменная

Событийная модель IE предоставляет информацию о событии в объекте Event, но IE никогда не передает объекты Event обработчикам событий в виде аргументов. Вместо этого IE делает объект Event доступным в виде свойства `event` глобального объекта Window. Это значит, что функция обработки события в IE может обращаться к объекту Event как к `window.event` или просто как к `event`. Использование глобальной переменной там, где годится аргумент функции, может показаться странным, но схема IE работает, так как в событийной модели программирования неявно предполагается, что одновременно всегда обрабатывается только одно событие. Поскольку два события никогда не обрабатываются одновременно, можно спокойно сохранять информацию о текущем обрабатываемом событии в глобальной переменной.

Объект Event представляет собой глобальную переменную, и это несовместимо со стандартной событийной моделью DOM Level 2. Обойти это препятствие можно с помощью одной строки кода. Для того чтобы функция обработки события работала в обеих объектных моделях, напишите ее так, чтобы она ожидала аргумент, а затем, если аргумент не передан, инициализируйте этот аргумент из глобальной переменной. Например:

```
function portableEventHandler(e) {
    if (!e) e = window.event;  // Получение деталей события для IE

    // Тело обработчика событий
}
```

19.3.3. Всплывание события в IE

В событийной модели IE, в отличие от модели DOM Level 2, нет понятия перехвата события. Однако в модели IE – так же, как в модели Level 2, события всплывают по иерархии включения. Как и в модели Level 2, всплывание события относится только к необработанным событиям или событиям ввода (в основном к событиям мыши и клавиатуры), но не к высокоуровневым се-

мантическим событиям. Основное различие между всплыванием события в IE и событийной моделью DOM Level 2 состоит в способе остановки всплывания. В объекте Event в IE, в отличие от объекта Event в DOM, нет метода `stopPropagation()`. Чтобы предотвратить всплывание события или остановить всплывание далее вверх по иерархии включения, обработчик события в IE должен установить свойство `cancelBubble` объекта Event в `true`:

```
window.event.cancelBubble = true;
```

Обратите внимание, свойство `cancelBubble` применяется только к текущему событию. Когда генерируется новое событие, `window.event` присваивается новое событие Event, а `cancelBubble` принимает значение по умолчанию – `false`.

19.3.4. Регистрация обработчика события в IE

В IE 4 обработчики событий регистрируются так же, как и в исходной событийной модели Level 0: указанием их в виде атрибутов HTML или присваиванием функций свойствам-обработчикам событий элементов документа. Единственное отличие состоит в том, что IE 4 обеспечивает доступ и регистрацию обработчиков событий для всех элементов документа, а не только для элементов форм, изображений и ссылок, доступных в DOM Level 0.

IE 5 и более поздние версии вводят методы `attachEvent()` и `detachEvent()`, предоставляющие способ регистрации нескольких функций-обработчиков для данного типа события в данном объекте. Эти методы работают так же, как `addEventListener()` и `removeEventListener()`, за исключением того, что поскольку событийная модель IE не поддерживает перехват событий, они требуют только двух аргументов: типа события и функции-обработчика. Кроме того, в отличие от событийной модели Level 2, имена обработчиков событий, передаваемых методу IE, должны включать префикс «on»: «onclick», а не «click». Вот так обработчик события регистрируется с помощью метода `attachEvent()`:

```
function highlight() { /* Здесь находится код обработчика события */ }
document.getElementById("myelt").attachEvent("onmouseover", highlight);
```

Другое различие между `attachEvent()` и `addEventListener()` состоит в том, что функции, зарегистрированные с помощью `attachEvent()`, вызываются как глобальные функции, а не как методы элемента документа, в котором произошло событие. То есть, когда исполняется обработчик события, зарегистрированный с помощью `attachEvent()`, ключевое слово `this` ссылается на объект Window, а не на целевой элемент события.

19.3.5. Пример: буксировка с использованием событийной модели IE

Пример 19.3 – это модифицированная версия функции `beginDrag()`, представленной в примере 19.2. Эта версия включает код, позволяющий функции работать с событийной моделью IE, в дополнение к событийной модели

DOM Level 2. Назначение и предполагаемое использование этой версии beginDrag() совпадает с примером 19.2, поэтому если вы поняли тот пример, то легко разберетесь и в этом. Интересным этот пример делает то, что он помещает рядом две событийные модели, четко выделяя их отличия.

Самое большое отличие версии кода для IE состоит в том, что он должен полагаться на всплывание событий, а не на перехват. Обычно тут все гладко, но это не идеальное решение данной задачи. Другое важное отличие, на которое следует обратить внимание, – обработчикам событий в IE не передается объект Event. Заметьте, что код в этом примере также различает, с одной стороны, IE 5 и более поздние версии, поддерживающие attachEvent(), и с другой стороны, IE 4, не поддерживающие этот метод. Пример HTML-документа, использующего эту функцию beginDrag(), можно найти в обсуждении примера 19.2.

Пример 19.3. Буксировка с использованием событийной модели IE

```
/**
 * PortableDrag.js:
 * beginDrag() предназначена для вызова из обработчика события onmousedown.
 * elementToDrag может быть элементом, получившим событие mousedown, либо
 * каким-либо из содержащих его элементов. event должен быть объектом Event
 * для события mousedown. Эта реализация работает как с событийной моделью
 * DOM Level 2 так и с событийной моделью IE.
 **/
function beginDrag(elementToDrag, event) {
    // Вычисляем расстояние между левым верхним углом элемента и местом
    // нажатия кнопки мыши. Эти значения потребуются функции moveHandler,
    // определенной ниже.
    var deltaX = event.clientX - parseInt(elementToDrag.style.left);
    var deltaY = event.clientY - parseInt(elementToDrag.style.top);

    // Регистрируем обработчики, которые будут реагировать на события
    // mousemove и mouseup, следующие за событием mousedown.
    if (document.addEventListener) {  // DOM Level 2 Event Model
        // Регистрируем перехватывающие обработчики событий
        document.addEventListener("mousemove", moveHandler, true);
        document.addEventListener("mouseup", upHandler, true);
    }
    else if (document.attachEvent) {  // Событийная модель IE 5+
        // В событийной модели IE мы не можем перехватывать события, поэтому
        // эти обработчики вызываются только если событие всплывает к ним.
        // При этом предполагается, что нет никаких промежуточных элементов,
        // которые перехватывают события и не дают им всплывать.
        document.attachEvent("onmousemove", moveHandler);
        document.attachEvent("onmouseup", upHandler);
    }
    else {  // Событийная модель IE 4
        // В IE 4 мы не можем использовать attachEvent(), поэтому назначаем
        // обработчики событий напрямую после сохранения любых ранее
        // зарегистрированных обработчиков, чтобы они могли быть восстановлены.
```

```
    // Заметьте, что здесь мы тоже полагаемся на всплывание событий.
    var oldmovehandler = document.onmousemove;
    var olduphandler = document.onmouseup;
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}

// Мы обработали это событие. Не разрешаем кому-либо еще его увидеть.
if (event.stopPropagation) event.stopPropagation();  // DOM Level 2
else event.cancelBubble = true;                      // IE

// Предотвращаем любое действие по умолчанию.
if (event.preventDefault) event.preventDefault();   // DOM Level 2
else event.returnValue = false;                     // IE

/**
 * Это обработчик, который перехватывает события mousemove во время
 * перетаскивания элемента. Он отвечает за перемещение элемента.
 **/
function moveHandler(e) {
    if (!e) e = window.event;  // Событийная модель IE

    // Перемещаем элемент к текущей позиции курсора мыши, по необходимости
    // подстраиваем позицию на смещение начального нажатия.
    elementToDrag.style.left = (e.clientX - deltaX) + "px";
    elementToDrag.style.top = (e.clientY - deltaY) + "px";

    // И не даем кому-либо еще увидеть это событие.
    if (e.stopPropagation) e.stopPropagation();  // DOM Level 2
    else e.cancelBubble = true;                  // IE
}

/**
 * Это обработчик, перехватывающий финальное событие mouseup, которое
 * возникает в конце перетаскивания.
 **/
function upHandler(e) {
    if (!e) e = window.event;  // IE Event Model

    // Отменяем регистрацию перехватывающих обработчиков событий
    if (document.removeEventListener) {  // Событийная модель DOM
        document.removeEventListener("mouseup", upHandler, true);
        document.removeEventListener("mousemove", moveHandler, true);
    }
    else if (document.detachEvent) {  // Событийная модель IE 5+
        document.detachEvent("onmouseup", upHandler);
        document.detachEvent("onmousemove", moveHandler);
    }
    else {  // Событийная модель IE 4
        document.onmouseup = olduphandler;
        document.onmousemove = oldmovehandler;
    }
```

```
            // И не даем событию распространяться дальше.
            if (e.stopPropagation) e.stopPropagation();  // DOM Level 2
            else e.cancelBubble = true;                  // IE
        }
    }
```

19.4. Событийная модель Netscape 4

Событийная модель Netscape 4 похожа на исходную событийную модель Level 0 за исключением того, что она предоставляет информацию о событиях в объекте Event, передаваемом в качестве аргумента функции-обработчику. Она также поддерживает специальные методы для выполнения перехвата событий. Эти возможности объясняются в следующих разделах.

19.4.1. Объект Event в Netscape 4

Событийная модель Netscape 4 определяет объект Event, содержащий информацию о произошедшем событии. Как и модель DOM Level 2, она передает объект Event в качестве аргумента всем обработчикам событий. Однако почти все свойства объекта Event в Netscape 4 абсолютно не совпадают со свойствами объекта Event в IE и объектов событий DOM Level 2. Вот эти свойства:

`type`

Строка, указывающая тип произошедшего события. Она совпадает с именем обработчика события без префикса «on» (например, «click» или «mousedown»). Это свойство совместимо с объектами Event в IE и DOM Level 2.

`target`

Элемент документа, в котором произошло событие. Это свойство совместимо со свойством `target` объекта Event DOM Level 2 и сравнимо с `srcElement` объекта Event в Internet Explorer.

`pageX`, `pageY`

Эти свойства указывают координаты (в пикселах), по которым произошло событие, относительно левого верхнего угла окна. Для документов, размер которых превышает размер окна, надо прибавить смещения `window.pageXOffset` и `window.pageYOffset` для преобразования значений свойства в координаты документа. Сравнимо со свойствами `clientX` и `clientY` объекта MouseEvent в DOM Level 2 и объекта Event в IE.

`which`

Целое, указывающее, какая кнопка мыши или клавиша была нажата. Для событий мыши левая, средняя и правая кнопки обозначаются, соответственно, значениями 1, 2 и 3. Сравните это со свойствами button (взаимно несовместимыми) объекта MouseEvent в DOM Level 2 и объекта Event в IE. Для клавиатурных событий это свойство содержит код Unicode нажатой клавиши. Сравните это со свойством `keyCode` объекта Event в IE.

`modifiers`

Целое, указывающее, какие клавиши-модификаторы на клавиатуре были нажаты, когда произошло событие. Значение представляет собой битовую маску, составленную из следующих значений: `Event.ALT_MASK`, `Event.CONTROL_MASK`, `Event.META_MASK` и `Event.SHIFT_MASK`. Сравнимо со свойствами `altKey`, `ctrlKey`, `metaKey` и `shiftKey` объекта MouseEvent DOM Level 2 и объекта Event в IE.

В событийной модели Netscape 4 объект Event передается всем обработчикам событий. Когда обработчик определен в виде строки кода JavaScript в атрибуте HTML, этот код неявно преобразуется в функцию с аргументом `event`. Это значит, что обработчики событий HTML могут ссылаться на объект Event с помощью идентификатора `event`. (Сравните с моделью IE, в которой идентификатор `event` ссылается на глобальный объект Event. Реализации совершенно различны, но практический результат одинаков.)

Для обратной совместимости объекты Event в Mozilla и Netscape 6 реализуют большинство свойств объекта Event Netscape 4, за примечательным исключением (на момент написания этой книги) свойства `modifiers`.

19.4.2. Перехват событий в Netscape 4

Событийная модель Netscape 4 не поддерживает всплывание событий, как модель IE, но поддерживает в ограниченной форме их перехват, как модель DOM Level 2. (Фактически модель распространения событий стандарта DOM – это комбинация модели перехвата Netscape и модели всплывания IE.) Netscape 4 поддерживает некоторую форму перехвата событий, но этот перехват значительно отличается от определенного событийной моделью DOM Level 2.

В Netscape 4 объекты Window, Document и Layer могут запросить возможность предварительного просмотра событий до обработки сгенерировавшими их элементами. Такой запрос делается вызовом метода `captureEvents()` этих объектов. Аргумент этого метода задает тип перехватываемых событий; это битовая маска, составленная из констант, определенных как статические свойства конструктора Event. Поэтому, например, если программа хочет, чтобы все события `mousedown` и `mouseup` перехватывались объектом Window до их обработки объектами, для которых они предназначались, она может вызвать `captureEvents()` следующим образом:

```
window.captureEvents(Event.MOUSEDOWN | Event.MOUSEUP);
```

Сделав такой запрос на получение событий, программа затем должна зарегистрировать обработчики для этих событий:

```
window.onmousedown = function(event) { ... };
window.onmouseup = function(event) { ... };
```

Когда один из этих обработчиков событий перехватывает событие, он должен решить, что делать с ним дальше. В некоторых программах перехваченное событие обрабатывается и дальше не распространяется. Однако иногда программе требуется передать событие дальше. Если вы передаете событие методу `routeEvent()` объектов Window, Document и Layer, метод передает со-

бытие следующему объекту Window, Document или Layer, который воспользовался captureEvents() для перехвата этого типа событий. Или, если других перехватывающих объектов, куда может быть направлено событие, нет, оно направляется в свой первоначальный объект-источник и вызывается соответствующий обработчик события этого объекта. Например:

```
function clickHandler(event) {
  if (event.which == 3) {  // Это правая кнопка мыши
    // Обрабатываем событие здесь и больше ничего не делаем
    // Событие дальше не распространяется
  }
  else {  // Это не правая кнопка мыши
    // Это событие нас не интересует, поэтому направляем его
    // в какой-либо другой элемент, который в нем заинтересован
    window.routeEvent(event);
  }
}
```

Альтернативой вызову routeEvent() является передача объекта Event в метод handleEvent() объекта, которому требуется передать событие. Метод handleEvent() передает событие соответствующему обработчику этого объекта.

Если объект Window, Document или Layer больше не желает перехватывать события, он должен вызвать метод releaseEvents(), указав тот же аргумент, который передан captureEvents().

Модели перехвата событий Netscape 4 и DOM Level 2 фундаментально несовместимы. Так, модель DOM по умолчанию распространяет перехваченные события, а модель Nescape – нет. Mozilla и Netscape 6 реализуют API перехвата событий Netscape 4, но этот API, как оказалось, не работает.

19.4.3. Пример: буксировка в событийной модели Netscape 4

Пример 19.4 – это реализация знакомого нам метода beginDrag() в событийной модели Netscape 4 (и DOM Netscape 4 на основе слоев). Он показывает, как перехватываются события и как пишутся обработчики для этой модели. Этот пример включает и код JavaScript, и простой документ HTML, который посредством метода beginDrag() определяет изображение, которое пользователь может перетащить. Сравните эту реализацию метода beginDrag() с двумя рассмотренными ранее и определявшими свои вложенные функции обработки событий в начале функции beginDrag(), а не в ее конце. Это сделано для обхода ошибки: вложенные функции, помещенные в конец функции beginDrag(), в Netscape 4 не работают. Также обратите внимание на обработчик onmousedown в конце этого примера: он разрешает буксировку, только если нажата клавиша Shift, и проверяет нажатие этого модификатора средствами API объекта Event Netscape 4, существенно отличающегося от API DOM Level 2 и IE.[1]

[1] На момент написания этой книги Mozilla и Netscape 6 не обеспечивали совместимость со свойством modifiers объекта Event Netscape 4, поэтому показанный здесь обработчик onmousedown работает только в Netscape 4, но не в Netscape 6.

Пример 19.4. Буксировка в Netscape 4

```
<script>
/**
 * Эта функция предназначена для использования в обработчике событий mousedown
 * объекта на слое. Первый аргумент должен быть объектом Layer. Второй аргумент
 * должен быть объектом Event для события mousedown.
 **/
function beginDrag(layerToDrag, event) {
    // Эта вложенная функция реагирует на события mousemove и передвигает слой.
    function moveHandler(event) {
        // Перемещаем элемент к текущей позиции курсора мыши, по необходимости
        // подстраиваем позицию на смещение начального нажатия.
        layerToDrag.moveTo(event.pageX - deltaX, event.pageY-deltaY);

 // Никаких действий по умолчанию не выполняем, далее событие не распространяем.
        return false;
    }

    // Эта вложенная функция обрабатывает события mouseup.
    // Она останавливает перехват событий и отменяет регистрацию обработчиков.
    function upHandler(event) {
        // Прекращаем перехват и обработку событий перетаскивания
        document.releaseEvents(Event.MOUSEMOVE | Event.MOUSEUP);
        document.onmousemove = null;
        document.onmouseup = null;

 // Никаких действий по умолчанию не выполняем, далее событие не распространяем.
        return false;
    }

    // Вычисляем расстояние между левым верхним углом слоя и местом нажатия
    // кнопки мыши. Эти значения потребуются затем для функции moveHandler.
    var deltaX = event.pageX - layerToDrag.left;
    var deltaY = event.pageY - layerToDrag.top;

    // Организуем перехват событий mousemove и mouseup.
    // Затем организуем их обработку с помощью определенных выше функций.
    document.captureEvents(Event.MOUSEMOVE | Event.MOUSEUP);
    document.onmousemove = moveHandler;
    document.onmouseup = upHandler;
}
</script>
<!-- Вот как может использоваться beginDrag() в Netscape 4 -->
<!-- Определяем слой с помощью CSS-атрибутов -->
<div id="div1" style="position:absolute; left:100px; top:100px;">
<!-- Помещаем на слой некоторое содержимое и обработчик -->
<!-- события mousedown -->
<img src="plus.gif" width="20" height="20"
     onmousedown="if (event.modifiers & Event.SHIFT_MASK)
                      beginDrag(window.document.div1, event);">
</div>
```

20

Приемы обеспечения совместимости

JavaScript, как и Java, является представителем нового поколения языков, независимых от платформы. Иначе говоря, можно написать программу на JavaScript и предполагать, что она без изменений будет работать в поддерживающем JavaScript броузере, запущенном на любом типе компьютера с любым типом операционной системы. Однако это идеал, а мы живем в несовершенном мире и пока не достигли такого совершенства.

Есть, и видимо всегда будут существовать проблемы совместимости, которые программисты JavaScript должны иметь в виду. Не следует забывать одно обстоятельство – нас окружает неоднородная среда. Программы JavaScript могут работать в трех и более операционных системах, на трех и более версиях броузера от как минимум двух поставщиков. Тем из нас, кто пришел из непереносимого прошлого, когда программы разрабатывались для конкретной платформы, учитывать это бывает сложно. Помните: платформа, на которой разрабатывается программа, не имеет значения. Программа может прекрасно работать на этой платформе, но для того чтобы по-настоящему проверить программу, надо выяснить, работает ли она (или изящно отказывает) на всех платформах, для которых предназначена.

Вопросы совместимости делятся на две обширных категории: с одной стороны, относящиеся к специфическим возможностям платформы, броузера и версии; а с другой стороны, ошибки и несовместимость на уровне языка, включая несовместимость JavaScript с не поддерживающими JavaScript броузерами. В этой главе обсуждаются приемы борьбы с несовместимостями обеих категорий. Если вы прошли через все предыдущие главы этой книги, вы, вероятно, эксперт в JavaScript, и можете писать серьезные программы на JavaScript. Однако не публикуйте свои программы в Интернете (или в неоднородной внутренней сети), пока не прочитаете эту главу!

20.1. Совместимость с платформами и броузерами

При разработке качественного кода JavaScript тестирование и знание несовместимостей, специфических для платформ, марок и версий, – ваши главные помощники. Например, тот, кто знает, что Netscape 2 на платформах Macintosh всегда получает время, отличное от настоящего примерно на час, может предпринять шаги для решения этой проблемы. Тот, кто знает, что Netscape 2 и 3 на платформах Windows автоматически не сбрасывают настройки строки состояния при перемещении курсора мыши из гиперссылки, может написать соответствующий обработчик события, явно очищающий строку состояния. Тот, кто знает, что Internet Explorer 4 и Netscape 4 поддерживают совершенно различные модели Dynamic HTML, может создавать страницы, использующие нужный механизм в зависимости от типа броузера.

Знание существующих несовместимостей является ключевым для написания совместимого кода. К сожалению, создание точного списка всех известных несовместимостей по маркам, версиям и платформам – задача титаническая. Она не ставится в рамках этой книги и, видимо, никогда всерьез не решалась. Некоторую помощь можно найти в Интернете, но в основном вам придется полагаться на собственный опыт и тестирование. Однако есть несколько базовых подходов, призванных помочь тем, кто столкнется однажды с несовместимостью. Это подходы описаны в следующих разделах.

20.1.1. Подход «наименьшего общего знаменателя»

Одна из технологий решения вопросов несовместимости – избегать их, как чумы. Например, объект Date в Netscape 2 известен наличием ошибок. Если программист хочет, чтобы пользователи Netscape 2 могли работать с его программой, он может вообще избегать применения объекта Date.[1]

Другой пример: Netscape 3 и IE 3 оба поддерживают свойство `opener` объекта Window, а Netscape 2 его не поддерживает. Подход «наименьшего общего знаменателя» предписывает отказаться от этого свойства, если необходимо сохранить совместимость с Netscape 2. Вместо этого можно при открытии нового окна создавать эквивалентное собственное свойство:

```
newwin = window.open("", "new", "width=500, height=300");
newwin.creator = self;
```

Если устанавливать свойство `creator` последовательно для каждого нового создаваемого окна, то можно полагаться на это свойство, а не на непереносимое свойство `opener`. (Как мы увидим позднее, другая альтернатива заключается в отказе от совместимости с Netscape 2 и обязательном наличии броузе-

[1] Вообще-то я не рекомендую это делать. На момент написания данной книги Netscape 2 настолько устарел, что его можно спокойно игнорировать.

ра, поддерживающего JavaScript 1.1 или более позднюю версию, поскольку все такие броузеры поддерживают свойство `opener`.)

Благодаря этому приему вы используете только те возможности, о которых известно, что они работают на всех целевых платформах. Он не позволяет писать самые современные программы, но приводит к созданию переносимых, безопасных программ, которые могут выполнять многие полезные функции.

20.1.2. Защитное кодирование

Подход под названием «защитное кодирование» подразумевает написание кода, содержащего независимые от платформы обходные пути решения проблем совместимости платформ. Например, если для отображения специального сообщения в строке состояния вы устанавливаете свойство `status` объекта Window из обработчика события `onmouseover`, строка состояния очищается при перемещении курсора мыши из гиперссылки, за исключением Windows-версий Netscape 2 и 3. Для того чтобы обойти это препятствие, возьмите за правило включать в программу обработчик события `onmouseout` для очистки строки состояния. Эта предосторожность исправляет ошибку в текущих (и будущих) платформах, в которых она имеется, и не наносит никакого вреда на платформах, ее не имеющих.

20.1.3. Проверка наличия возможностей

Проверка наличия возможностей – полезная технология борьбы с несовместимостью. Если вам нужна какая-то возможность, которая может поддерживаться не во всех броузерах, включите в сценарий код, проверяющий, поддерживается ли данная возможность. Если требуемая возможность на текущей платформе не поддерживается, либо откажитесь от нее на этой платформе, либо напишите альтернативный код, работающий на всех платформах.

Вернемся к свойству `opener`. В подходе «наименьший общий знаменатель» мы просто обошлись без него и предусмотрели для всех платформ альтернативный вариант. В случае проверки наличия возможностей мы предоставляем альтернативу, только когда текущая платформа не поддерживает `opener`:

```
newwin = window.open("", "new", "width=500, height=300");
if (!newwin.opener) newwin.opener = self;
```

Теперь мы проверяем существование свойства `opener`. Та же технология работает для проверки существования методов. Например, метод `split()` объекта String реализован только в JavaScript 1.1. Мы можем написать собственную версию этой функции, работающую во всех версиях JavaScript, но для эффективности предпочли бы быстрый встроенный метод на тех платформах, которые его поддерживают. Следовательно, наш код проверки возможности применения метода `split()` может получиться таким:

```
if (s.split)              // Проверяем существование метода без его вызова
    a = s.split(":");     // Если он существует, мы можем его вызвать
else                      // В противном случае:
    a = myspliit(s, ":");  // используем нашу альтернативную реализацию
```

Проверка наличия возможности часто применяется для создания эффектов DHTML, поддерживаемых только на некоторых броузерах или реализуемых в разных броузерах по-разному. Например, разрабатывая сайт, содержащий эффекты переключения изображений, можно реализовать проверку наличия данной возможности с помощью следующего кода:

```
if (document.images) {  // Если броузер определяет массив images[],
                        // мы включаем здесь код переключения изображений
}
// В противном случае просто опускаем реализацию этого эффекта
```

И еще один пример: предположим, что мы хотим работать с динамически позиционируемым элементом документа. В различных броузерах имеются для этого различные API, поэтому сначала проверим наличие возможности, чтобы определить, какой API поддерживается данным броузером:

```
if (document.getElementById) {  // Если поддерживается W3C DOM API,
    // выполняем DHTML-эффект с использованием W3C DOM API
}
else if (document.all) {        // Если поддерживается IE 4 API,
    // выполняем DHTML-эффект с использованием IE 4 API
}
else if (document.layers) {     // Если поддерживается Netscape 4 API,
    // выполняем DHTML-эффект (насколько это возможно), используя
    // Netscape 4 API
}
else {                          // В противном случае, DHTML не поддерживается,
    // и предоставляем, если это возможно, статическую альтернативу DHTML
}
```

Преимущество технологии проверки наличия возможности состоит в том, что она дает код, не привязанный к конкретному списку разработчиков и номеров версий броузеров. Эта технология работает с существующим на сегодня набором броузеров, и будет продолжать работать с будущими броузерами, какие бы возможности они ни реализовывали.

20.1.4. Обходные решения для конкретных платформ

Проверка наличия возможности хорошо подходит для больших функциональных областей. Например, эта технология позволяет определить, поддерживает ли броузер переключение изображений или W3C DOM API. С другой стороны, иногда приходится обходить отдельные ошибки и особенности определенного броузера, и простого пути проверки наличия ошибки может не быть. В таких случаях приходится создавать специфическое для

платформы обходное решение, привязанное к определенному броузеру, его версии и операционной системе (или некоторой их комбинации).

Вспомните, что свойство navigator объекта Window предоставляет информацию о марке и версии броузера и операционной системы, под управлением которой он работает. Руководствуясь этой информацией, можно вставить в программу код, специфический для платформы.

В качестве примера специфического для платформы обходного решения упомянем свойство bgColor объекта Document. На платформах Windows и Macintosh можно установить это свойство во время выполнения для изменения цвета фона документа. К сожалению, при этом в Unix-версиях Netscape 2 и 3 цвет изменяется, но содержимое документа временно исчезает. Если вы хотите создать спецэффект, основанный на изменении цвета фона, то с помощью объекта navigator можете проверить, не на платформе ли Unix запущен ли броузер, и просто опустить код спецэффекта для этих платформ. Код может выглядеть следующим образом:

```
// Проверяем, что мы работаем в Netscape 2 или 3 на платформе Unix
var nobg = (parseInt(navigator.appVersion) < 4) &&           // Версия
           (navigator.appName.indexOf("Netscape") != -1) &&  // Марка
           (navigator.appVersion.indexOf("X11") != -1);      // ОС

// Если нет, идем дальше и выполняем анимацию цвета фона
if (!nobg) animate_bg_color();
```

При написании обходного кода, специфического для платформ, часто используется код клиентских модулей проверки окружения («снифферов») для определения текущей платформы на основе (обычно) свойств объекта navigator. Вы один раз запускаете код модуля проверки окружения, и он устанавливает переменные, описывающие текущую платформу. После этого уже не надо анализировать свойства объекта navigator в каждом специфическом для платформы фрагменте написанного вами кода; можно просто использовать переменные, установленные модулем проверки. Простой модуль проверки окружения, возможностей которого достаточно для многих задач, может выглядеть, например, так:

```
var browserVersion = parseInt(navigator.appVersion);
var isNetscape = navigator.appName.indexOf("Netscape") != -1;
var isIE = navigator.appName.indexOf("Microsoft") != -1;
var agent = navigator.userAgent.toLowerCase();
var isWindows = agent.indexOf("win") != -1;
var isMac = agent.indexOf("mac") != -1;
var isUnix = agent.indexOf("X11") != -1;
```

Определив переменные подобным образом, можно написать следующий код:

```
if (isNetscape && browserVersion < 4 && isUnix) {
    // Здесь можно обойти ошибку в Netscape 3 на платформах Unix
}
```

В Интернете доступно множество готовых клиентских модулей проверки окружения. Обширную коллекцию (с полезным обсуждением их применения) можно найти по адресу *http://www.mozilla.org/docs/ web-developer/sniffer/browser_type.html*.

20.1.5. Обеспечение совместимости при помощи серверных сценариев

Еще один специфический для платформы подход к обеспечению совместимости возможен, если веб-приложение использует сценарии на стороне сервера, например сценарии CGI или серверный JavaScript. Программа на стороне сервера может проанализировать поле `User-Agent` заголовка HTTP-запроса, которое позволяет определить, какой броузер запущен у пользователя. Имея эту информацию, программа может генерировать специализированный код JavaScript, о котором известно, что он правильно работает в этом броузере. Или, если серверный сценарий обнаруживает, что броузер пользователя не поддерживает JavaScript, он может генерировать веб-страницы, основанные вообще не на JavaScript. Серьезный недостаток этого подхода в том, что серверный сценарий не может выявить ситуацию, когда пользователь отключил поддержку JavaScript в своем броузере.

Заметим, что вопросы программирования CGI и серверных сценариев вообще выходят за рамки этой книги.

20.1.6. Игнорировать проблему

При рассмотрении любых вопросов несовместимости важно понимать, насколько это важно. Если несовместимость небольшая или косметическая, касается броузера или платформы, которые не распространены широко, или только устаревшей версии броузера, то можно просто игнорировать проблему, и пусть пользователи, которых это касается, борются с ней сами.

Например, ранее я предлагал определить обработчик события `onmouseout`, исправляющий некорректную очистку строки состояния в Netscape 2 и 3 для Windows. К сожалению, обработчик события `onmouseout` не поддерживается в Netscape 2, поэтому такое обходное решение не будет работать для этой платформы. Если у программиста есть основания полагать, что у его приложения будет много пользователей, работающих с Netscape 2 для Windows, и он думает, что очистка строки состояния действительно важна, то ему придется найти какое-нибудь другое решение. В собственном обработчике событий `onmouseover` можно было бы посредством `setTimeout()` организовать очистку строки состояния каждые две секунды. Но это решение не идеально: что если курсор мыши все еще находится над гиперссылкой и строка состояния не должна очищаться в данный момент? В этом случае проще игнорировать проблему. Такой подход вполне оправдан, поскольку Netscape 2 уже давно можно считать устаревшим, а любых пользователей, все еще применяющих его, следует побуждать к обновлению программного обеспечения.

20.1.7. Изящный отказ

И наконец, имеются несовместимости, которые не могут игнорироваться и для которых нет обходных решений. В этих случаях программа должна работать корректно на всех платформах, броузерах и версиях, которые предоставляют необходимую функциональность, и выполнять «изящный отказ» на всех других. При «изящном отказе» (fail gracefully) программа должна распознать отсутствие функциональности и выдать пользователю сообщение о невозможности работы с программой.

Например, технология замены изображений, рассмотренная в главе 14, не работает в Netscape 2 или Internet Explorer 3, и не существует обходного решения, имитирующего эту технологию. Следовательно, мы не должны даже пытаться запустить программу на этих платформах, а должны вежливо сообщить пользователю о несовместимости.

Реализовать изящный отказ иногда труднее. Остаток этой главы в значительной степени объясняет, как это сделать.

20.2. Совместимость версий языка

В предыдущем разделе рассмотрены общие приемы обеспечения совместимости, пригодные для различных версий броузеров, работающих на различных платформах. В этом разделе мы обращаемся к другому вопросу совместимости: как применять новые возможности JavaScript, чтобы не возникали ошибки на броузерах, не поддерживающих эти возможности. Наши цели просты: код JavaScript не должен интерпретироваться броузерами, не понимающими его, и должны выводиться специальные сообщения, информирующие пользователей о том, что их броузеры не могут запускать данные сценарии.

20.2.1. Атрибут language

Первая цель достигается просто. Как мы видели в главе 12, можно предотвратить попытки броузера запускать код, который он не понимает, установкой нужного значения атрибута `language` в теге `<script>`. Например, следующий тег `<script>` указывает, что содержащийся в нем код ориентирован на возможности JavaScript 1.1 и что броузеры, не поддерживающие эту версию языка, не должны пытаться его запускать:

```
<script language="JavaScript1.1">
    // код JavaScript 1.1
</script>
```

Обратите внимание, что применение атрибута `language` – универсальный прием. Будучи установленным в «JavaScript1.2», этот атрибут предотвращает попытки броузеров с версиями JavaScript 1.0 или 1.1 запускать данный код. На момент написания этой книги последние версии броузеров (Netscape 6 и IE 6) поддерживают версии языка 1.0, 1.1, 1.2, 1.3, 1.4 и 1.5.

Если, например, код JavaScript содержит инструкцию `try/catch`, то для обработки исключений необходимо поместить его в тег `<script>` с `language="JavaScript1.5"`, чтобы броузеры, не понимающие эту инструкцию, не пытались его запускать.

К сожалению, функциональность атрибута `language` искажена. Дело в том, что указание `language="JavaScript1.2"` делает поведение броузера Netscape несовместимым со стандартом ECMA-262. Например, как мы видели в главе 5, установка атрибута `language` равным этому значению приводит к тому, что оператор `==` проверяет равенство без выполнения каких-либо преобразований типов. Как мы видели в главе 8, указание значения «JavaScript1.2» также приводит к тому, что метод `toString()` ведет себя совсем по-другому. Если только вы не хотите, чтобы это новое, несовместимое поведение было реализовано, или если вы тщательно не обходите все эти несовместимости, то должны избегать `language="JavaScript1.2"`.

Обратите внимание, что номера версий, задаваемые в атрибуте `language`, соответствуют номерам версий интерпретатора JavaScript от Netscape (и теперь от Mozilla). Интерпретатор от Microsoft более или менее следовал эволюции интерпретатора от Netscape, но имейте в виду, что атрибут `language` в некоторой степени специфичен для марки броузера: нет гарантии совпадения возможностей языка, поддерживаемых различными марками броузеров для данного номера версии. Это особенно касается значения `language="JavaScript1.2"`, но стоит соблюдать осторожность и для других версий. К сожалению, нет способа указать версию спецификации с помощью атрибута `language`. То есть *нельзя* написать:

```
<script language="ECMAScript3">...</script>
```

20.2.2. Явная проверка версии

Атрибут `language` позволяет частично решить проблему совместимости версий языка, но только частично. Надо также иметь возможность выполнять изящный отказ в броузерах, не поддерживающих требуемую версию JavaScript. Если нам требуется JavaScript 1.1, мы бы хотели иметь возможность уведомить пользователей броузеров с JavaScript 1.0, что они не могут просматривать эту страницу. Пример 20.1 показывает, как это сделать.

Пример 20.1. Сообщение для броузеров, не поддерживающих JavaScript 1.1

```
<!-- Устанавливаем переменную, определяющую, какую версию JavaScript мы -->
<!-- поддерживаем. Эта технология может быть расширена до любого количества  -->
<!-- версий                                                                  -->
<script language="JavaScript"> var _version = 1.0; </script>
<script language="JavaScript1.1">  _version = 1.1; </script>
<script language="JavaScript1.2">  _version = 1.2; </script>

<!-- Запускаем этот код на любом броузере, поддерживающем JavaScript. -->
<!-- Если номер версии недостаточен, выводим сообщение -->
<script language="JavaScript">
```

```
  if (_version < 1.1) {
    document.write('<hr><h1>Эта страница требует JavaScript 1.1</h1>');
    document.write('Ваш броузер JavaScript 1.0 не может запускать эту страницу.');
    document.write('<hr>');
  }
</script>

<!-- Теперь запускаем саму программу только на броузерах JavaScript 1.1 -->
<script language="JavaScript1.1">
    // Здесь указан фактический код JavaScript 1.1
</script>
```

20.2.3. Подавление ошибок, связанных с версиями

В примере 20.1 приведен код JavaScript 1.1, который не пытаются исполнять броузеры JavaScript 1.0. А что если мы хотим писать код JavaScript 1.2, который не пытаются исполнять броузеры JavaScript 1.1? Можно в атрибуте `language` явно указать «JavaScript1.2», но, как мы обсуждали ранее, это приводит к несовместимому поведению Netscape. К сожалению, JavaScript 1.2 добавляет в язык много нового синтаксиса. Если в коде есть инструкция `switch`, инициализатор объекта или функциональный литерал, то в результате запуска этого кода в броузере JavaScript 1.1 будут получены синтаксические ошибки времени выполнения.

Один из способов обхода этой проблемы состоит в подавлении любых ошибок, возникающих в броузерах JavaScript 1.1. В примере 20.2 показывается, как это сделать при помощи обработчика ошибок `onerror` объекта Window (который был описан в главе 13).

Пример 20.2. Подавление ошибок, связанных с версией

```
<!-- Проверка, поддерживается ли JavaScript 1.2 -->
<script language="JavaScript1.2">var _js12_ = 1.2</script>

<!-- Избегаем проблем с JavaScript 1.2 в Netscape, запуская этот код -->
<!-- на любом броузере, поддерживающем JavaScript 1.1. Если, однако,  -->
<!-- броузер не поддерживает JavaScript 1.2, мы отображаем сообщение -->
<!-- об ошибке и подавляем любые возникшие синтаксические ошибки. -->
<script language="JavaScript1.1">
// Если JavaScript 1.2 не поддерживается, изящно отказываем
function suppressErrors() { return true; }
if (!_js12_) {
    window.onerror = supressErrors;
    alert("Эта программа требует броузера с поддержкой JavaScript 1.2 ");
}

// Продолжаем код JavaScript 1.2
</script>
```

20.2.4. Загрузка новой страницы для обеспечения совместимости

Еще один подход к совместимости версий состоит в загрузке веб-страницы, требующей определенного уровня поддержки JavaScript, после того как выяснено, предоставляет ли броузер этот уровень поддержки. В примере 20.3 показано, как это сделать с помощью короткого сценария, проверяющего, поддерживается ли JavaScript 1.2. Если броузер поддерживает эту версию, сценарий вызывает метод `Location.replace()` для загрузки новой веб-страницы, требующей JavaScript 1.2. Если JavaScript 1.2 не поддерживается, сценарий выводит соответствующее сообщение.

Пример 20.3. Веб-страница для проверки на совместимость JavaScript

```
<head>
<script language="JavaScript1.2">
// Если JavaScript 1.2 поддерживается, извлекаем новый URL из части нашего
// URL после вопросительного знака, и загружаем новый URL
location.replace(location.search.substring(1));

// Входим в долгий пустой цикл, чтобы тело этого документа не отображалось,
// пока загружается новый документ
for(var i = 0; i < 10000000; i++);
</script>
</head>
<body>
<hr size="4">
<h1>Эта страница требует JavaScript 1.2</h1>
Ваш броузер не может запустить эту страницу. Обновите его
на поддерживающий JavaScript 1.2, например, на Netscape 4
или на Internet Explorer 4
<hr size="4">
</body>
```

Самое интересное в этом примере, что он универсальный – имя загружаемого файла JavaScript 1.2 кодируется в исходном URL, и этот файл загружается, только если поддерживается JavaScript 1.2. Другими словами, если имя файла в этом примере *testjs12.html*, то можно поместить его в URL, показанный в следующей гиперссылке:

```
<a href="http://my.isp.net/~david/utils/testjs12.html?../js/cooljs12.html">
Посетите мою крутую страницу JavaScript 1.2!
</a>
```

Кроме того, надо сказать, что в примере 20.3 вызов `Location.replace()` начинает загрузку новой страницы, но не прекращает загрузку текущей страницы сразу. Следовательно, код JavaScript в этом примере после вызова `replace()` входит в длинный пустой цикл. Это позволяет избежать разбора и отображения оставшейся части документа, чтобы сообщение, предназначенное для

броузеров, не поддерживающих JavaScript 1.2, не отображалось в броузерах, поддерживающих эту версию.

И наконец, отметим, что прием, показанный в примере 20.3, помогает отличать не только одну версию JavaScript от другой, но и броузеры, поддерживающие и не поддерживающие JavaScript. В следующем разделе обсуждается еще один прием обеспечения совместимости, полезный для броузеров, которые не поддерживают JavaScript.

20.3. Совместимость с броузерами, не поддерживающими JavaScript

В предыдущем разделе обсуждалась совместимость между броузерами, не поддерживающими определенную версию JavaScript. Здесь же рассматривается совместимость с броузерами, вообще не поддерживающими JavaScript. Это либо броузеры, не имеющие средств поддержки JavaScript, либо броузеры, в которых пользователь отключил поддержку JavaScript (что некоторые делают по соображениям безопасности). Поскольку такие броузеры все еще встречаются, веб-страницы должны уметь изящно отказывать при открытии в броузере, не понимающем JavaScript. Действия, связанные с этим, делятся на две части: во-первых, необходимо позаботиться о том, чтобы код JavaScript не отображался в виде HTML-текста, а во-вторых, вы должны организовать вывод сообщения, информирующего посетителя, что его броузер не может корректно обрабатывать страницу.

20.3.1. Скрытие кода сценариев от старых броузеров

Веб-броузеры, поддерживающие JavaScript, исполняют инструкции, расположенные между тегами `<script>` и `</script>`. Броузеры, не поддерживающие JavaScript, но понимающие тег `<script>`, игнорируют все, что находится между `<script>` и `</script>`. Так и должно быть. Однако очень старые броузеры (и некоторые из них все еще используются) даже не понимают этих тегов. Это значит, что они игнорируют сами теги и рассматривают весь код JavaScript, расположенный между ними, как HTML-текст, который предназначен для отображения. Если не принять соответствующих мер, пользователи этих старых броузеров увидят код JavaScript, отформатированный в виде больших бессмысленных абзацев и представленный в виде содержимого веб-страницы!

Чтобы избежать этого, заключите тело сценария в HTML-комментарий, используя формат, показанный в примере 20.4.

Пример 20.4. Сценарий, скрытый от старых броузеров

```
<script language="JavaScript">
<!-- Начало HTML-комментария, скрывающего скрипт
        // Инструкции JavaScript
        //                .
```

```
        //                .
// Конец HTML-комментария, скрывающего скрипт -->
</script>
```

Броузеры, не понимающие теги <script> и </script>, просто их игнорируют. Следовательно, первая и седьмая строки в примере 20.4 не действуют на эти броузеры. Они также игнорируют строки со второй по шестую, поскольку первые четыре символа во второй строке обозначают начало HTML-комментария, а последние три символа в шестой строке обозначают его конец, – все, что находится между ними, игнорируется анализатором HTML.

Технология скрытия сценариев также действует в броузерах, *поддерживающих* JavaScript. Первая и седьмая строки указывают на начало и конец сценария. Интерпретаторы клиентского JavaScript распознают открывающую строку HTML-комментария <!--, но рассматривают ее как однострочный комментарий. Следовательно, броузер с поддержкой JavaScript рассматривает вторую строку как однострочный комментарий. Аналогично, шестая строка начинается со строки однострочного комментария //, поэтому она также игнорируется броузерами, поддерживающими JavaScript. Остаются строки с третьей по пятую, исполняемые как инструкции JavaScript.

Да, к этому надо немного привыкнуть, но это простое и элегантное смешение комментариев HTML и JavaScript дает именно то, что нам нужно, – код JavaScript не отображается броузерами, не поддерживающими JavaScript. Количество броузеров, нуждающихся в таком комментировании, уменьшается, но его все еще можно встретить в коде JavaScript в Интернете. Конечно, комментарии не обязательно должны быть такими многословными, как в примере 20.4. Часто встречаются такие сценарии:

```
<script language="JavaScript">
<!--
  document.write(new Date());
// -->
</script>
```

Такой способ комментирования позволяет скрыть код JavaScript от броузеров, которые не могут его исполнить. Следующий шаг изящного отказа заключается в выводе сообщения о том, что страницу запустить невозможно.

20.3.2. <noscript>

Теги <noscript> и </noscript> заключают произвольный блок кода HTML, который должен отображаться любым броузером, не поддерживающим JavaScript. Это позволяет сообщить пользователю, что его броузер не может корректно отобразить страницы, требующие поддержки JavaScript. Например:

```
<script language="JavaScript1.1">
  // Ваш код JavaScript
</script>
<noscript>
```

```
<hr size="4">
<h1>Эта страница требует JavaScript 1.1</h1>
Эта страница требует броузера, поддерживающего JavaScript 1.1.<p>
Ваш броузер либо не поддерживает JavaScript, либо поддержка JavaScript
в нем отключена. Если вы хотите корректно видеть эту страницу, пожалуйста,
обновите ваш броузер или включите поддержку JavaScript.
<hr size="4">
</noscript>
```

Тег `<noscript>` порождает одну неприятность. Он был введен компанией Netscape в броузере Netscape 3. Следовательно, он не поддерживается в Netscape 2. А поскольку Netscape 2 не поддерживает `<noscript>` и `</noscript>`, он игнорирует эти теги и отображает текст, указанный между ними, даже несмотря на то, что поддерживает сценарии. Но в предыдущем примере это нам на руку, так как мы указываем, что коду требуется поддержка JavaScript 1.1.

21

Безопасность в JavaScript

По причине открытой природы Интернета безопасность – вопрос серьезный. Это стало особенно актуальным с появлением языков, подобных Java и JavaScript, так как они допускают встраивание исполняемого содержимого в статичные веб-страницы. Поскольку загрузка веб-страницы может привести к исполнению произвольного кода на компьютере, требуются строгие меры безопасности для предотвращения повреждения данных или нарушения секретности злонамеренным кодом. В этой главе обсуждаются вопросы безопасности, относящиеся к JavaScript. Обратите внимание, что здесь *не* обсуждаются многие другие вопросы, относящиеся в безопасности в Интернете, например технологии аутентификации и криптографии, призванные обеспечить защиту содержимого веб-документов и HTML-форм при их передаче по сети.

21.1. JavaScript и безопасность

Вот первая линия обороны JavaScript против злонамеренного кода – язык просто не поддерживает некоторые возможностей. Например, клиентский JavaScript не предоставляет никакого способа записи или удаления файлов и каталогов на клиентском компьютере. Без объекта File и функций доступа к файлам программа JavaScript не может удалять данные или разводить вирусы в пользовательской системе.

Аналогично, клиентский JavaScript не имеет никаких сетевых примитивов. Программа на JavaScript может загружать URL и отсылать данные HTML-форм веб-серверам и сценариям CGI или отправлять их по электронной почте, но не может устанавливать прямое соединение с какими-либо другими узлами сети. Это, например, означает, что программа на JavaScript не может использовать машину клиента как платформу для атаки, с нее пытаясь

взломать пароли на другой машине. (Это было бы особенно опасно, если бы программа, загруженная из Интернета через брандмауэр, могла пытаться внедриться во внутреннюю сеть, защищенную этим брандмауэром.)

Базовый язык JavaScript и базовая объектная модель на стороне клиента не имеют средств работы с файловой системой и сетевых средств, необходимых для большей части злонамеренного кода, но ситуация не так проста, как кажется. Во многих веб-броузерах JavaScript используется как «машина сценариев» для других программных компонентов, таких как управляющие элементы ActiveX в Internet Explorer и надстройки в Netscape. Эти компоненты могут иметь файловые и сетевые возможности, и тот факт, что программы JavaScript могут управлять ими, омрачает картину и вновь поднимает вопросы безопасности. Особенно это касается управляющих элементов ActiveX, и Microsoft приходилось выпускать исправления, относящиеся к системе безопасности, чтобы запретить коду JavaScript использовать возможности объектов ActiveX. Мы кратко остановимся на этом вопросе снова в конце главы.

Умышленное исключение некоторых возможностей обеспечивает базовый уровень защиты от наиболее очевидных атак, оно не решает других проблем безопасности. Речь идет в основном о секретности – программам JavaScript не должен быть разрешен экспорт информации о пользователе броузера, если предполагается, что эта информация секретна.

Когда пользователь просматривает веб-страницы, он по умолчанию согласен передать информацию, в частности, о том, какой броузер запущен на его компьютере. Как стандартная часть протокола HTTP с каждым запросом веб-страницы передается строка, идентифицирующая броузер, его версию и производителя. Эта информация относится к разряду открытой, как, например, IP-адрес вашего интернет-соединения. Но есть и данные, которые не должны быть открытыми, в том числе адрес электронной почты, который не должен распространяться, если только пользователь не отправляет сообщение по электронной почте или не разрешает автоматическую отправку сообщения электронной почты от своего имени.

Должна обеспечиваться и секретность истории просмотра страниц (записи о том, какие сайты посещал пользователь) и секретность содержимого списка закладок. История просмотра и закладки говорят многое об интересах пользователя. За эту информацию специалисты по маркетингу (и не только) платят большие деньги, чтобы повысить эффективность своих рассылок с предложениями о распродажах. Можете быть уверены, что если ваш броузер или JavaScript позволяют украсть эту ценную частную информацию, найдутся люди, которые будут красть ее каждый раз, когда вы посещаете их сайты, и через несколько секунд она уже будет на рынке. Большинство веб-пользователей, знающих, что любой сайт, который они посещают, может узнать, что они интересуются, например, кошками, а также женской обувью и клубом Sierra, будут чувствовать себя неловко.

Даже если предположить, что у нас нет странных увлечений, которые мы хотели бы скрыть, есть много других серьезных причин, по которым следует

беспокоиться о секретности данных. Одна из них – утилитарная забота о сохранении тайны. Мы не хотим, чтобы программа на JavaScript, загруженная из Интернета и работающая в одном окне броузера, могла изучать содержимое других окон броузера, содержащих страницы из корпоративной сети, находящейся за брандмауэром. Далее в этой главе рассказывается, как JavaScript защищен от таких злоупотреблений.

21.2. Ограничения в JavaScript

Вы уже знаете, что первая линия обороны против злонамеренных сценариев в клиентском JavaScript – отсутствие в языке некоторых возможностей. Вторая линия обороны – ограничения, налагаемые JavaScript на некоторые возможности, им поддерживаемые. Например, клиентский JavaScript поддерживает метод `close()` для объекта Window, но большинство (будем надеяться, что все) реализаций веб-броузеров ограничивают этот метод так, что сценарий может закрыть только окно, которое было открыто сценарием с того же веб-сервера. В частности, сценарий не может закрыть окно, открытое пользователем; если он пытается это сделать, то пользователь видит окно, содержащее просьбу подтвердить, действительно ли он хочет закрыть окно.

Самое важное из этих ограничений безопасности известно как *политика общего происхождения* (*same-origin policy*) и описано в следующем разделе. Ниже приведен список других ограничений, имеющихся в большинстве реализаций клиентского JavaScript. Это не исчерпывающий список. Каждый броузер может иметь несколько отличный набор ограничений, и собственные возможности каждого броузера вполне могут иметь дополнительные фирменные ограничения по безопасности.

- Объект History изначально был разработан как массив URL, представляющих полную историю просмотра страниц в броузере. Однако когда связанные с секретностью следствия этой возможности стали очевидны, в объекте History были оставлены только методы `back()`, `forward()` и `go()` для перемещения броузера по массиву History, не раскрывающие содержимое массива.

- Свойство `value` объекта FileUpload не может быть установлено. Если бы это свойство было доступно, сценарий мог бы установить его значение равным любому желаемому имени файла и заставить форму загрузить на сервер содержимое любого указанного файла (например, файла паролей).

- Сценарий не может отправить содержимое формы (с помощью, например, метода `submit()` объекта Form) по URL `mailto:` или `news:` без явного подтверждения пользователя через диалоговое окно. Такая передача данных может содержать адрес электронной почты пользователя, не подлежащий распространению без разрешения пользователя.

- Программа на JavaScript не может закрыть окно броузера без подтверждения пользователя, если только программа сама не открыла это окно.

Это не дает злонамеренным сценариям вызвать метод `self.close()`, чтобы закрыть окно броузера и выйти из программы.

- Сценарий не может открыть окно, размеры которого по одной стороне меньше 100 пикселов, или изменить размеры окна, чтобы они стали меньше 100 пикселов по одной стороне. Аналогично, такой сценарий не может переместить окно за пределы экрана или создать окно, превышающее размеры экрана. Это не дает сценариям открывать окна, которые пользователь не может видеть или легко может не заметить; такие окна могут содержать сценарии, продолжающие работать после того, как пользователь решил, что они завершились. Кроме того, сценарий не может создавать окна без заголовка, потому что такое окно может имитировать системное диалоговое окно и обманом заставить пользователя ввести, например, секретный пароль.
- Сценарий не может заставить окно или фрейм отображать URL `about:`, например `about:cache`, так как эти URL могут открывать системную информацию, например, содержимое кэша броузера.
- Сценарий не может устанавливать какие-либо свойства объекта Event. Это не дает сценариям имитировать события. Пользователь не может регистрировать в документах слушатели событий или перехватывать события для документов, загруженных из источников, отличных от источника данного сценария. Это не дает сценариям подсматривать данные (например, нажатия клавиш, составляющие значение пароля), введенные пользователем на других страницах.

21.3. Политика общего происхождения

В JavaScript есть одно далеко идущее ограничение безопасности, заслуживающее отдельного раздела. Это ограничение известно как политика общего происхождения: сценарий может читать только свойства окон и документов, имеющих общее с самим сценарием происхождение (то есть загруженных с того же узла, через тот же порт и по тому же протоколу).

Политика общего происхождения на самом деле применяется не ко всем свойствам всех объектов в окне, имеющем другое происхождение. Но она применяется ко многим из них и, в частности, применяется практически ко всем свойствам объекта Document. В любом случае можете считать, что все предопределенные свойства всех клиентских объектов, имеющих другое происхождение, закрыты для ваших сценариев. Определенные пользователем свойства объектов, имеющих другое происхождение, также могут быть недоступны, хотя в каких-то реализациях это может быть и не так.

Политика общего происхождения – довольно серьезное ограничение, но оно необходимо для того, чтобы не допустить кражу внутренней информации. Без этого ограничения недоверенный сценарий (возможно, сценарий, загруженный в броузер в защищенной корпоративной сети через брандмауэр) в одном окне смог бы использовать методы DOM для чтения содержимого до-

кументов в других окнах броузера, которые могут содержать секретную информацию.

Тем не менее в некоторых ситуациях политика общего происхождения оказывается слишком строгой. Это создает особые проблемы для крупных веб-сайтов, которые могут использовать несколько серверов. Например, сценарий с *home.netscape.com* может законно читать свойства документа, загруженного с *developer.netscape.com*, а сценариям с *orders.acme.com* может потребоваться прочитать свойства из документов с *catalog.acme.com*. Чтобы поддерживать такие крупные веб-сайты, в JavaScript 1.1 вводится свойство `domain` объекта Document. По умолчанию свойство `domain` содержит имя сервера, с которого был загружен документ. Это свойство можно установить только равным строке, являющейся допустимым доменным суффиксом первоначального значения. Другими словами, если значение `domain` первоначально равно строке «home.netscape.com», то можно установить его равным «netscape.com», но не «home.netscape» или «cape.com», и конечно не «microsoft.com». (Значение `domain` должно содержать по крайней мере одну точку; нельзя установить его равным «com» или другому домену верхнего уровня.)

Если два окна (или фрейма) содержат сценарии, установившие одинаковые значения свойства `domain`, политика общего происхождения для этих двух окон ослабляется, и каждое из окон может читать значения свойств другого окна. Например, взаимодействующие сценарии в документах, загруженных с *orders.acme.com* и с *catalog.acme.com*, могут установить свойства `document.domain` равными «acme.com», тем самым указывая общее происхождение документов и разрешая каждому из документов читать свойства другого.

21.4. Зоны безопасности и подписанные сценарии

«Безразмерная» политика безопасности никогда не может удовлетворять целиком. Если политика слишком строга, доверенные сценарии не имеют возможность делать интересные и полезные вещи, которых мы бы от них хотели. С другой стороны, если политика слишком терпима, недоверенные сценарии могут привести к разорению! Идеальным решением является возможность такой настройки политики безопасности, при которой доверенные сценарии были бы предметом меньшего количества ограничений, чем недоверенные. Два основных поставщика броузеров, Microsoft и Netscape, взяли за основу различные подходы к обеспечению настраиваемой безопасности, кратко описанные в этом разделе.

Internet Explorer определяет «зоны безопасности», где пользователь может перечислить веб-сайты, сценариям которых он доверяет, и веб-сайты, сценариям которых он не доверяет. Кроме того, можно по отдельности настроить политики безопасности для этих двух зон, давая больше прав и накладывая меньше ограничений на доверенные сайты. (Можно также по отдельности настраивать полномочия интернет- и интранет-сайтов, явно не перечисленных ни в одной из двух других зон.)

К сожалению, это решение нельзя назвать ни полным ни глубоко проработанным, поскольку большинство параметров безопасности, которые разрешается настраивать в IE, напрямую не относятся к JavaScript. Например, в бета-версии IE 6 можно указать, разрешено ли сценариям управлять объектами ActiveX и апплетами Java, и могут ли они выполнять операции вставки (cut and paste). Но нет, например, средства, позволяющего отключить политику общего происхождения для доверенного сайта или разрешить сценариям с доверенных сайтов отправлять сообщения электронной почты без запроса подтверждения пользователя.

Netscape 4 и Netscape 6 реализуют настройку безопасности с помощью подхода, известного как «подписанные сценарии». Подписанные сценарии предоставляют полную избирательную настройку политик безопасности и делают это криптографически безопасным и теоретически очень стойким образом. К сожалению, эта многообещающая технология никогда не была популярной. Дело в том, что Microsoft не имеет совместимой технологии, поэтому процесс создания подписанных сценариев оказывается обременительным для авторов, а применение подписанных сценариев может быть непонятным для конечных пользователей.

Если говорить коротко, то подписанный сценарий содержит неподделываемую цифровую подпись, указывающую организацию или лицо, написавшее сценарий или другим образом принимающее ответственность за него. Когда подписанному сценарию требуется обойти одно из описанных выше ограничений безопасности, он сначала запрашивает специальные привилегии, разрешающие ему это делать. Когда сценарий запрашивает привилегии, броузер обращается к пользователю, сообщая, кем подписан сценарий, и спрашивая, хочет ли он выдать запрошенные полномочия сценарию, написанному данным лицом или организацией. Приняв решение, пользователь может указать броузеру запомнить свой выбор, чтобы в будущем этот вопрос ему не задавался. Благодаря такой процедуре пользователь может формировать избирательную настраиваемую политику безопасности на лету, по мере возникновения необходимости.

Как уже говорилось, процесс создания подписанных сценариев несколько обременителен. Кроме того, технология их создания изменилась между версиями Netscape 4 и Netscape 6. Эта тема выходит за рамки данной книги, но подробную информацию можно найти по адресам *http://developer.netscape.com/docs/manuals/signedobj/trust/index.htm* и *http://www.mozilla.org/projects/security/components/*.

22

Совместное применение Java и JavaScript

В главе 14 мы уже говорили, что Netscape 3 и более поздние версии, а также Internet Explorer 3 и более поздние версии разрешают программам JavaScript читать и записывать открытые поля и вызывать открытые методы апплетов Java, встроенных в HTML-документы. Netscape обеспечивает взаимодействие JavaScript с апплетами Java посредством технологии, известной как LiveConnect. Internet Explorer, со своей стороны, рассматривает любой объект Java (включая апплеты) как элемент управления ActiveX и использует технологию сценариев ActiveX, позволяющую программам на JavaScript взаимодействовать с Java. Так как технология Netscape специально предназначена для связи JavaScript и Java, она имеет возможности, которые не предоставляет технология ActiveX в IE. Однако на практике эти две технологии вполне сравнимы. Материал этой главы базируется на Netscape LiveConnect, но ключевые возможности, описанные в ней, работают и в IE.[1]

В начале этой главы обсуждается применение JavaScript в сценариях апплетов Java, вызов кода JavaScript апплетами Java и то, как (только в Netscape) JavaScript позволяет работать непосредственно с системными классами Java. Затем описаны важные детали работы LiveConnect. Здесь предполагается, что вы имеете по крайней мере базовое представление о программировании на Java (см. книги «Java in a Nutshell»[2] Дэвида Флэнагана (David Flanagan) и «Learning Java» (Изучаем Java) Патрика Нимейера (Patrick Niemeyer) и Джонатана Кнудсена (Jonathan Knudsen), O'Reilly).

---

[1] Обратите внимание, что в Netscape 6 поддержка LiveConnect была слабой, но в Netscape 6.1 и более поздних версиях она реализована полностью.

[2] Дэвид Флэнаган «Java. Справочник». – Пер. с англ. – СПб: Символ-Плюс, 2004.

22.1. Применение Java-апплетов в сценариях

В главе 14 говорилось, что апплеты Java, встроенные в веб-страницу, становятся элементами массива Document.applets[]. А если в апплете указан атрибут name или id, то к апплету можно обращаться напрямую как к свойству объекта Document. Например, к апплету, созданному с помощью тега <applet> с атрибутом name, равным «chart», можно обратиться как к document.chart.

Открытые поля и методы любого апплета доступны JavaScript, как если бы они были свойствами и методами объекта JavaScript. Так, если апплет с именем chart определяет поле с именем lineColor, имеющее тип String, программа JavaScript может получать и устанавливать значение этого поля так:

```
var chartcolor = document.chart.lineColor;  // Чтение поля апплета
document.chart.lineColor = "#ff00ff";       // Установка поля апплета
```

JavaScript может даже читать и устанавливать значения полей, являющихся массивами. Предположим, что апплет chart определяет два метода, объявленные следующим образом (код Java):

```
public int numPoints;
public double[] points;
```

Программа JavaScript может использовать эти поля так:

```
for(var i = 0; i < document.chart.numPoints; i++)
    document.chart.points[i] = i*i;
```

Этот пример иллюстрирует сложный момент, относящийся к связи между Java и JavaScript: преобразование типов. Java – это строго типизированный язык с большим количеством отдельных элементарных типов. JavaScript слабо типизирован и имеет только один числовой тип. В предыдущем примере целое (integer) Java преобразуется в число JavaScript, а различные числа JavaScript – в значения типа double в Java. Чтобы эти значения были нужным образом преобразованы, выполняется очень много закулисной работы. Далее в этой главе преобразование типов данных рассматривается подробно.

Кроме получения и установки значений полей апплета Java, JavaScript может также вызывать методы апплета. Предположим, что апплет chart определяет метод с именем redraw(). Этот метод не требует аргументов и просто уведомляет апплет о том, что массив points[] был изменен, и апплет должен выполнить перерисовку. JavaScript может вызывать этот метод так же, как если бы это был метод JavaScript:

```
document.chart.redraw();
```

JavaScript может вызывать методы, принимающие аргументы и возвращающие значения. LiveConnect или технология сценариев ActiveX преобразуют значения аргументов JavaScript в допустимые значения Java, а возвращаемые значения – в допустимые значения JavaScript. Предположим, что апплет chart определяет следующие методы Java:

```
public void setDomain(double xmin, double xmax);
public void setChartTitle(String title);
public String getXAxisLabel();
```

Из JavaScript эти методы вызываются так:

```
document.chart.setDomain(0, 20);
document.chart.setChartTitle("y = x*x");
var label = document.chart.getXAxisLabel();
```

И наконец, методы Java могут возвращать объекты Java, а JavaScript может получать и устанавливать значения открытых полей и вызывать открытые методы этих объектов. Предположим, что апплет Java определяет метод с именем `getXAxis()`, возвращающий объект Java, который является экземпляром класса Axis, и метод `setYAxis()`, принимающий тот же тип в качестве аргумента. Теперь предположим, что в классе Axis есть метод `setTitle()`. Мы можем использовать эти методы в таком коде JavaScript:

```
var xaxis = document.chart.getXAxis();  // Получаем объект Axis
var newyaxis = xaxis.clone();           // Делаем его копию
newyaxis.setTitle("Y");                 // Вызываем метод объекта...
document.chart.setYAxis(newyaxis);      // и передаем его другому методу
```

В случае применения JavaScript для вызова методов объекта Java есть одна сложность. Java допускает существование нескольких методов с одинаковым именем, если они имеют различные типы аргументов. Например, объект Java может объявить следующие два метода:

```
public String convert(int i);     // Преобразует целое в строку
public String convert(double d);  // Преобразует число с плавающей точкой
```

JavaScript имеет только один числовой тип и не различает целые значения и значения с плавающей точкой, поэтому когда вы с помощью JavaScript передаете число методу с именем «convert», интерпретатор не может понять, какой метод вы собираетесь вызвать. На практике эта проблема возникает редко, и ее обычно можно обойти, просто переименовав методы. Последние версии LiveConnect (в Netscape 6.1 и более поздних версиях) также позволяют устранить такие неоднозначности, включив типы аргумента в имя метода. Так, если два метода, приведенные выше, определены в `document.applets[0]`, их можно отличить следующим образом:

```
var iconvert = document.applets[0]["convert(int)"];
                                        // Получаем целочисленный метод
iconvert(3);  // Вызываем метод
```

22.2. Применение JavaScript из Java

Узнав, как управлять кодом Java из JavaScript, мы можем перейти к противоположной задаче: управлению кодом JavaScript из Java. Такое управление в основном осуществляется через класс Java *netscape.javascript.JSObject*,

представляющий объект JavaScript внутри программы на Java. Возможности связи JavaScript с Java, описанные в предыдущем разделе, обычно прекрасно работают и в Netscape и в Internet Explorer. В отличие от этого, приемы связи Java с JavaScript, описанные здесь, поддерживаются не так хорошо, и ошибки встречаются как в Netscape, так и в IE.

22.2.1. Класс JSObject

Любое взаимодействие Java с JavaScript выполняется через экземпляр класса *netscape.javascript.JSObject*. Экземпляр этого класса представляет собой обертку вокруг одиночного объекта JavaScript. Класс определяет методы, позволяющие читать и писать значения свойств и элементы массива объекта JavaScript, а также вызывать методы объекта. Вот описание этого класса:

```
public final class JSObject extends Object {
    // Статический метод для получения объекта JSObject для окна броузера,
    // в котором содержится объект.
    public static JSObject getWindow(java.applet.Applet applet);
    public Object getMember(String name);              // Чтение свойства объекта
    public Object getSlot(int index);                  // Чтение элемента массива
    public void setMember(String name, Object value);  // Установка свойства объекта
    public void setSlot(int index, Object value);      // Установка элемента массива
    public void removeMember(String name);                // Удаление свойства
    public Object call(String methodName, Object args[]); // Вызов метода
    public Object eval(String s);                         // Вычисление строки
    public String toString();                            // Преобразование в строку
    protected void finalize();
}
```

Все объекты JavaScript образуют иерархию, вершиной которой является текущее окно броузера, поэтому объекты JSObject должны также образовывать иерархию. Для того чтобы взаимодействовать с любыми объектами JavaScript, апплет Java должен сначала получить объект JSObject, представляющий окно броузера (или фрейм), в котором находится апплет. В классе JSObject не определен метод-конструктор, поэтому мы не можем просто создать нужный нам JSObject. Вместо этого необходимо вызвать статический метод `getWindow()`. Метод, которому передается ссылка на апплет, возвращает JSObject, представляющий окно броузера, которое содержит апплет. Следовательно, любой апплет, взаимодействующий с JavaScript, включает строку, которая выглядит примерно так:

```
JSObject jsroot = JSObject.getWindow(this);  // "this" - это сам апплет
```

Получив JSObject, ссылающийся на корневое окно в иерархии объектов JavaScript, можно посредством методов экземпляра JSObject читать значения свойств объекта JavaScript, который он представляет. Большинство этих свойств имеют значения, которые сами являются объектами JavaScript, поэтому можно продолжить процесс и также прочитать их свойства. Метод `getMember()` объекта JSObject возвращает значение свойства с данным име-

нем, а метод getSlot() – значение элемента массива с данным номером указанного объекта JavaScript. Вот как эти методы применяются:

```
import netscape.javascript.JSObject;  // Это должно быть в начале файла
    ...
JSObject jsroot = JSObject.getWindow(this);                    // self
JSObject document = (JSObject) jsroot.getMember("document");  //  .document
JSObject applets = (JSObject) document.getMember("applets");  //    .applets
Applet applet0 = (Applet) applets.getSlot(0);                 //      [0]
```

Здесь следует обратить внимание на два момента. Во-первых, методы getMember() и getSlot() оба возвращают значение типа «Object», которое, как правило, должно быть преобразовано в некоторое более конкретное значение, такое как JSObject. Во-вторых, значение, прочитанное из слота 0 массива applets, может быть преобразовано в Applet, а не в JSObject. Дело в том, что элементы массива applets[] в JavaScript – это объекты JavaScript, представляющие объекты Applet из Java. Когда Java читает объект JavaObject из JavaScript, обертка этого объекта «снимается» и возвращается объект Java, содержащийся в ней (в данном случае Applet). Преобразование данных, происходящее через интерфейс JSObject, описывается далее в этой главе.

Класс JSObject также поддерживает методы для установки значений свойств и элементов массивов объектов JavaScript. Методы setMember() и setSlot() соответствуют методам getMember() и getSlot(). Они устанавливают значение свойства с данным именем или элемента массива с указанным номером. Заметьте, однако, что устанавливаемое значение должно быть объектом Java. Значение элементарного типа устанавливается с помощью соответствующего класса-обертки Java: например, используется объект Integer вместо значения типа int. И наконец, метод removeMember() позволяет удалить значение указанного свойства из объекта JavaScript.

Кроме чтения и установки значений свойств и элементов массивов из объектов JavaScript класс JSObject позволяет вызывать методы объектов JavaScript. Метод call() класса JSObject вызывает указанный метод заданного объекта JavaScript и передает массив объектов Java в качестве аргументов этого метода. Как мы видели при установке значений свойств JavaScript, невозможно передать методу JavaScript в качестве аргументов значения элементарных типов Java; тут нужны соответствующие объектные типы Java. Например, посредством метода call() можно открыть новое окно броузера:

```
public JSObject newwin(String url, String window_name)
{
    Object[] args = { url, window_name };
    JSObject win = JSObject.getWindow(this);
    return (JSObject) win.call("open", args);
}
```

В классе JSObject имеется еще один важный метод: eval(). Этот метод Java работает так же, как функция JavaScript с тем же именем – он исполняет строку, содержащую код JavaScript. Работать с eval() часто намного проще,

чем с другими методами класса JSObject. Код передается в виде строки, поэтому можно использовать строковые представления нужных типов данных и не надо преобразовывать элементарные типы Java в соответствующие им объектные типы. Сравните, например, следующие две строки кода, устанавливающие свойства главного окна броузера:

```
jsroot.setMember("i", new Integer(0));
jsroot.eval("self.i = 0");
```

Вторая строка более понятна. Как еще один пример рассмотрим применение `eval()` для записи в определенный фрейм, отображаемый в окне броузера:

```
JSObject jsroot = JSObject.getWindow(this);
jsroot.eval("parent.frames[1].document.write('Привет от Java!')");
```

Сделать то же самое без метода `eval()` намного сложнее:

```
JSObject jsroot = JSObject.getWindow(this);
JSObject parent = (JSObject) jsroot.getMember("parent");
JSObject frames = (JSObject) parent.getMember("frames");
JSObject frame1 = (JSObject) frames.getSlot(1);
JSObject document = (JSObject) frame1.getMember("document");
Object[] args = { "Hello from Java!" };
document.call("write", args);
```

22.2.2. Применение JSObject в апплетах

Пример 22.1 иллюстрирует применение метода `init()` апплета, взаимодействующего с JavaScript при помощи LiveConnect.

Пример 22.1. Использование JavaScript из метода апплета

```
import netscape.javascript.*

public void init()
{
    // Получение JSObject, представляющего окно броузера, открываемое апплетом
    JSObject win = JSObject.getWindow(this);

    // Запускаем код  JavaScript с помощью eval().
    // Будьте осторожны с вложенными кавычками!
    win.eval("alert('Апплет CPUHog теперь работает на вашем компьютере. " +
       "Вы можете заметить, что ваша программа несколько замедлилась.');");
}
```

Для того чтобы можно было запускать любой апплет, его необходимо скомпилировать, а затем встроить в HTML-файл. Когда апплет взаимодействует с JavaScript, для выполнения обоих шагов требуются особые инструкции.

22.2.2.1. Компиляция апплетов, использующих класс JSObject

Любой апплет, взаимодействующий с JavaScript, использует класс *netscape.javascript.JSObject*. Следовательно, для того чтобы скомпилировать такой апплет, компилятор Java должен знать, где найти определение этого класса. Поскольку класс определен и поставляется Netscape, а не Sun, компилятор *javac* от Sun о нем не знает. Здесь рассказано, как заставить компилятор найти этот нужный класс. Тем, кто не работает с Sun JDK, надо обратиться за подробностями к документации от поставщика компилятора или среды разработки Java, так как им придется выполнять немного другие действия.

Чтобы сообщить компилятору JDK, где искать классы, надо установить переменную окружения `CLASSPATH`. Эта переменная окружения задает список каталогов и JAR-файлов (или ZIP-файлов), в которых компилятор должен искать определения классов (в дополнение к стандартному каталогу системных классов). Сложность состоит в том, чтобы выяснить, в каком файле в системе находится определение класса *netscape.javascript.JSObject*. Для Netscape 6.1 это файл *plugins/java2/javaplugin.jar* внутри каталога установки Netscape, а для Netscape 4 – файл *java/classes/java40.jar* в каталоге установки Netscape. Так, в Netscape 4 на платформе Windows файл *java40.jar*, скорее всего, находится в *C:\Program Files\Netscape\Communicator\Program\Java\Classes\java40.jar*.

Для Internet Explorer искомое определение класса обычно находится в одном из ZIP-файлов в каталоге *c:\Windows\Java\Packages*. Неприятность в том, что здесь находится группа файлов, имена которых совершенно невнятны и изменяются от версии к версии. Как правило, нужен самый большой из них. Убедиться, что он содержит файл *netscape/javascript/JSObject.class*, нетрудно, – например при помощи утилиты unzip.

Найдя нужный JAR- или ZIP-файл, можно сообщить компилятору об этом, установив переменную среды `CLASSPATH`. Для систем Unix установите ее так:

```
setenv CLASSPATH .:/usr/local/netscape/plugins/java2/javaplugin.jar
```

А для Windows так:

```
set CLASSPATH=.;C:\Windows\Java\Packages\5fpnnz7t.zip
```

Установив `CLASSPATH`, можно компилировать апплет с помощью *javac*.

22.2.2.2. Атрибут mayscript

Есть еще одно требование для запуска апплета, взаимодействующего с JavaScript. В качестве меры безопасности апплету не разрешается использовать JavaScript, если только автор веб-страницы (который может не быть автором апплета) явно не даст это разрешение (для чего необходимо включить новый атрибут `mayscript` в тег `<applet>` в HTML-файле).

В примере 22.1 показан фрагмент апплета, использующего JavaScript для вывода диалогового окна с предупреждением. Успешно скомпилировав этот апплет, можно включить его в HTML-файл следующим образом:

```
<applet code="CPUHog.class" width="300" height="300" mayscript></applet>
```

Без атрибута `mayscript` апплету нельзя будет использовать класс JSObject.

22.3. Непосредственное использование классов Java

Как говорилось в двух предыдущих разделах, и Netscape и Internet Explorer позволяют коду JavaScript взаимодействовать с апплетом Java, а апплету Java – взаимодействовать с JavaScript. Технология Netscape LiveConnect также позволяет программам на JavaScript создавать собственные объекты Java и использовать их, даже в отсутствие каких-либо апплетов. Internet Explorer не имеет какой-либо аналогичной возможности.

В Netscape объект `Packages` предоставляет доступ ко всем пакетам Java, о которых знает Netscape. Выражение `Packages.java.lang` ссылается на пакет *java.lang*, а выражение `Packages.java.lang.System` – на класс *java.lang.System*. Для удобства вместо `Packages.java` достаточно написать `java`. В Netscape код JavaScript может вызывать статический метод класса *java.lang.System* так:

```
// Вызов статического метода Java System.getProperty()
var javaVersion = java.lang.System.getProperty("java.version");
```

Такое применение не ограничено системными классами, поскольку LiveConnect позволяет создавать новые экземпляры классов Java посредством оператора JavaScript `new` (точно так же мы бы делали это в Java). В примере 22.2 показан код JavaScript, использующий стандартные классы Java (на самом деле код JavaScript выглядит практически так же, как код Java) для вывода окна и отображения в нем текста. Результат его работы показан на рис. 22.1.

Рис. 22.1. *Окно Java, созданное из JavaScript*

Пример 22.2. Использование встроенных Java-классов

```
var f = new java.awt.Frame("Hello World");
var ta = new java.awt.TextArea("hello, world", 5, 20);
f.add("Center", ta);
f.pack();
f.show();
```

Код этого примера создает простой пользовательский интерфейс Java. Однако в нем отсутствует какая-либо обработка событий или взаимодействие с пользователем. Подобная программа ограничена только выдачей результата, так как не содержит никакого способа уведомления кода JavaScript о взаимодействии пользователя с окном Java. Возможно, хотя и затруднительно, определить пользовательский интерфейс Java, реагирующий на события, средствами JavaScript. В Java 1.1 и более поздних версиях уведомление о событии выполняется путем вызова метода объекта EventListener. Апплеты Java могут исполнять произвольные строки кода JavaScript, поэтому можно определить класс Java, реализующий соответствующий интерфейс EventListener и вызывающий указанную строку кода JavaScript при уведомлении о возникновении события. Написав апплет с методом, позволяющим создавать такие объекты EventListener, можно средствами JavaScript формировать графические интерфейсы Java, включающие обработчики событий, определенные в JavaScript.

Заметим, что LiveConnect не дает полного доступа к Java-системе; другими словами, многое нельзя сделать с помощью LiveConnect. Например, LiveConnect не позволяет определять новые классы или подклассы Java из JavaScript, а также не дает возможности создавать массивы Java.[1] Кроме этих причин доступ к стандартным классам Java ограничен соображениями безопасности. Недоверенная программа на JavaScript не может работать, например, с классом *java.io.File*, так как это даст возможность читать, писать и удалять файлы системы. Недоверенный код JavaScript может использовать Java только теми способами, которыми это делают недоверенные апплеты.

22.4. Типы данных LiveConnect

Для того чтобы понять, что LiveConnect делает для соединения JavaScript с Java, необходимо разобраться в типах данных JavaScript, используемых в LiveConnect. Эти типы данных описаны в следующих разделах. Хотя в Internet Explorer реализована другая технология, понимание механизма работы LiveConnect поможет вам понять и работу IE. Некоторые из типов данных LiveConnect, описанные здесь, имеют аналоги в IE.

22.4.1. Класс JavaPackage

Пакет в Java представляет собой коллекцию связанных классов Java. Класс JavaPackage – это тип данных JavaScript, представляющий пакет Java. Свойства JavaPackage – это классы, которые содержатся в пакете (классы представляются классом JavaClass, который мы скоро рассмотрим), а также любые другие пакеты, которые содержатся в этом пакете. У класса JavaPackage есть ограничение: вы не можете использовать цикл `for/in` в Ja-

[1] Программы JavaScript могут создавать массивы опосредованно, с помощью метода `java.lang.reflect.Array.newInstance()` из Java 1.1.

vaScript для получения полного списка всех пакетов и классов, содержащихся в JavaPackage. Это ограничение является результатом внутреннего ограничения виртуальной машины Java.

Все объекты JavaPackage содержатся в родительском JavaPackage; свойство объекта Window с именем `Package` является объектом JavaPackage самого верхнего уровня, который служит вершиной этой иерархии пакетов. Он имеет такие свойства, как `java`, `sun` и `netscape`, которые являются объектами JavaScript, представляющими различные иерархии доступных броузеру классов Java. Например, объект JavaPackage `Package.java` содержит объект JavaPackage `Packages.java.awt`. Для удобства во всех объектах Window также имеются свойства `java`, `sun` и `netscape`, являющиеся сокращениями для `Packages.java`, `Packages.sun` и `Packages.netscape`. Следовательно, вместо `Packages.java.awt` вы можете набирать просто `java.awt`.

Продолжая пример, `java.awt` – это объект JavaPackage, содержащий объекты JavaClass, такие как `java.awt.Button`, который представляет класс `java.awt.Button`. Но в нем содержится другой объект JavaPackage, `java.awt.image`, представляющий пакет *java.awt.image* из Java.

Как можно видеть, схема назначения имен для свойств иерархии JavaPackage отражает схему назначения имен для пакетов Java. Однако обратите внимание, что имеется одно большое различие между классом JavaPackage и реальным пакетом Java, который он представляет. Пакеты в Java являются коллекциями классов, а не коллекциями других пакетов, то есть *java.lang* является именем пакета Java, а *java* не является. Поэтому объект JavaPackage с именем `java` фактически не представляет пакет Java – это просто удобное место в иерархии пакетов для других объектов JavaPackage, представляющих реальные пакеты Java.

На большинстве систем Java-классы устанавливаются в файлах в иерархии каталогов, соответствующих именам их пакетов. Например, класс *java.lang.String* хранится в файле *java/lang/String.class*. Фактически этот файл обычно содержится в ZIP-файле, но иерархия каталогов все равно сохраняется внутри архива. Следовательно, вместо того чтобы считать объект JavaPackage представлением пакета Java, вам может быть понятнее представлять его как каталог или подкаталог в иерархии каталогов классов Java.

Класс JavaPackage имеет несколько недостатков. В LiveConnect нет способа заранее определить, ссылается ли свойство JavaPackage на класс Java или на другой пакет Java, поэтому JavaScript предполагает, что это класс, и пытается загрузить класс. В результате, когда вы используете выражение вроде `java.awt`, LiveConnect сначала ищет файл класса *java/awt.class*. Он даже может искать этот класс в сети, приводя к возникновению ошибки «404 Файл не найден» на веб-сервере. Если LiveConnect не находит класс, он предполагает, что свойство ссылается на пакет, но нет способа убедиться, что пакет действительно существует, и в нем имеются реальные классы. Из этого следует еще один недостаток: если вы неправильно напишете имя класса, LiveConnect спокойно посчитает его именем пакета и не сообщит вам, что класс, который вы хотите использовать, не существует.

22.4.2. Класс JavaClass

Класс JavaClass – это тип данных JavaScript, представляющий класс Java. Объект JavaClass не имеет собственных свойств – все его свойства представляют (и имеют те же имена), что и открытые статические поля и методы представляемого им класса Java. Эти открытые статические поля и методы иногда называются *полями класса* и *методами класса*, указывая на их связь с классом, а не с экземпляром объекта. В отличие от класса JavaPackage, JavaClass допускает использование цикла `for/in` для перебора его свойств. Заметьте, что объект JavaClass не имеет свойств, представляющих поля и методы экземпляра класса Java – индивидуальные экземпляры класса Java представлены классом JavaObject, описанным в следующем разделе.

Как мы видели ранее, объекты JavaClass содержатся в объектах JavaPackage. Например, `java.lang` – это объект JavaPackage, в котором содержится свойство `System`. Следовательно, `java.lang.System` – это объект JavaClass, представляющий класс Java *java.lang.System*. Этот объект JavaClass, в свою очередь, имеет такие свойства, как `out` и `in`, представляющие статические поля класса *java.lang.System*. Подобным образом вы можете использовать JavaScript для обращения к любому из стандартных системных классов Java. Класс *java.lang.Double*, например, имеет имя `java.lang.Double` (или `Packages.java.lang.Double`), а класс *java.awt.Button* – `java.awt.Button`.

Другой способ получения объекта JavaClass в JavaScript – использование функции `getClass()`. Имея любой объект JavaObject, вы можете получить объект JavaClass, представляющий класс этого объекта Java, передав JavaObject в качестве аргумента функции `getClass()`.[1]

Имея объект JavaClass, вы можете использовать его несколькими способами. Класс JavaClass реализует функциональность LiveConnect, позволяющую программам JavaScript читать и устанавливать значения открытых статических полей классов Java и вызывать открытые статические методы классов Java. Например, `java.lang.System` – это объект JavaClass. Мы можем прочитать значение статического поля `java.lang.System` следующим образом:

```
var java_console = java.lang.System.out;
```

Аналогично можно вызвать статический метод класса `java.lang.System`:

```
var java_version = java.lang.System.getProperty("java.version");
```

Вспомните, что Java – типизированный язык – все поля и аргументы методов имеют тип. Если попытаться установить неправильное значение поля или передать аргумент неверного типа, генерируется исключение. (В версиях JavaScript до 1.5 возникает ошибка JavaScript.)

Класс JavaClass предоставляет еще одну важную возможность – использования объектов JavaClass с оператором `new` JavaScript для создания новых эк-

---

[1] Не путайте функцию `getClass()` JavaScript, возвращающую объект JavaClass, с методом `getClass()` из Java, возвращающим объект `java.lang.Class`.

земпляров классов Java – то есть для создания объектов JavaObject, что совпадает с синтаксисом создания объектов в JavaScript (и с синтаксисом Java):

```
var d = new java.lang.Double(1.23);
```

И наконец, создав подобным образом объект JavaObject, мы можем вернуться к функции getClass() и показать пример ее использования:

```
var d = new java.lang.Double(1.23);    // Создание JavaObject
var d_class = getClass(d);             // Получение JavaClass для JavaObject
if (d_class == java.lang.Double) ...;  // Результатом этого сравнения будет истина
```

Работая подобным образом со стандартными системными классами, мы, как правило, непосредственно используем имя системного класса, а не вызываем getClass(). Функция getClass() более полезна для получения класса пользовательского объекта, например экземпляра апплета.

Можно не обращаться к JavaClass с помощью длинных выражений вроде java.lang.Double, а определить переменную для упрощенного доступа к классу:

```
var Double = java.lang.Double;
```

Этим вы имитируете инструкцию import Java и можете увеличить эффективность вашей программы, так как LiveConnect не придется искать свойство lang в java и свойство Double в java.lang.

22.4.3. Класс JavaObject

Класс JavaObject – это тип данных JavaScript, представляющий объект Java. Классы JavaObject и JavaClass во многом похожи. Как JavaClass, JavaObject не имеет собственных свойств – все его свойства представляют (и имеют те же имена) открытые поля экземпляра и открытые методы экземпляра представляемого им объекта Java. Как и для JavaClass, в JavaScript можно в цикле for/in осуществить перебор всех свойств объекта JavaObject. Класс JavaObject реализует функциональность LiveConnect, позволяющую получать и устанавливать значения поля экземпляра и вызывать открытые методы объекта Java.

Пусть d – это JavaObject, представляющий экземпляр класса *java.lang.Double*, тогда метод этого объекта Java вызывается из кода JavaScript так:

```
n = d.doubleValue();
```

Аналогично, мы видели, что класс *java.lang.System* имеет статическое поле out, ссылающееся на объект класса Java *java.io.PrintStream*. В JavaScript на соответствующий объект JavaObject можно ссылаться следующим образом:

```
java.lang.System.out
```

и вызвать метод этого объекта:[1]

```
java.lang.System.out.println("Hello world!");
```

[1] Результат этой строки появляется не в веб-броузере, а в консоли Java. В Netscape 6, чтобы увидеть это окно, выберите Tasks → Tools → Java Console.

Объект JavaObject также позволяет читать и записывать значения открытых полей экземпляра представляемого им объекта Java. Однако ни класс *java.lang.Double*, ни класс *java.io.PrintStream*, использованные в предыдущих примерах, не имеют открытых полей экземпляра. Но предположим, что мы используем JavaScript для создания экземпляра класса *java.awt.Rectangle*:

```
r = new java.awt.Rectangle();
```

Затем мы можем прочитать и записать значения открытых свойств экземпляра с помощью следующего кода JavaScript:

```
r.x = r.y = 0;
r.width = 4;
r.height = 5;
var perimeter = 2*r.width + 2*r.height;
```

Изящество LiveConnect состоит в том, что он позволяет использовать объект Java `r` точно так же, как если бы это был объект JavaScript. Однако требуется некоторая осторожность: `r` – это JavaObject, и он не ведет себя точно так же, как обычные объекты JavaScript. Различия между ними будут описаны далее. Также помните, что в отличие от JavaScript, поля объектов Java и аргументы их методов являются типизированными. Если вы укажете значения JavaScript неверных типов, вы получите ошибки или исключения JavaScript.

В Netscape 6.1 и более поздних версиях класс JavaObject делает методы доступными по имени и по имени с указанием типа аргумента, что может быть полезно, когда имеется несколько методов, имеющих одинаковое имя, но требующих различные типы аргументов. Как мы видели ранее в этой главе, если JavaObject `o` обозначает объект, имеющий два метода с именем «convert», свойство `convert` объекта `o` может ссылаться на любой из этих методов. Однако в последних версиях LiveConnect объект `o` также определяет свойства, включающие типы аргументов, и вы можете указать, какая версия метода вам нужна, включив информацию о типах:

```
var iconvert = o["convert(int)"];  // Получаем требуемый метод
iconvert(3);                       // Вызываем его
```

Так как имя свойства включает скобки, вы не можете использовать для доступа к нему обычную нотацию с оператором «точка» (.) и должны указать его в виде строки в квадратных скобках. Тип JavaClass предоставляет такую возможность и для перегруженных статических методов.

22.4.4. Класс JavaArray

Последний тип данных LiveConnect для JavaScript – это класс JavaArray. Как вы можете предположить, экземпляры этого класса представляют массивы Java и предоставляют функциональность LiveConnect, позволяющую JavaScript читать элементы массивов Java. Как и массивы JavaScript (и массивы Java), объект JavaArray имеет свойство `length`, которое задает количество содержащихся в нем элементов. Элементы объекта JavaArray читаются с помощью стандартного оператора JavaScript для доступа к массиву – `[]`.

Кроме того, их можно перебрать с помощью цикла `for/in`. Вы можете использовать объекты JavaArray для доступа к многомерным массивам (фактически массивам массивов), как в JavaScript или в Java.

Предположим, что мы создаем экземпляр класса *java.awt.Polygon*:

```
p = new java.awt.Polygon();
```

В объекте JavaObject `p` имеются свойства `xpoints` и `ypoints`, являющиеся объектами JavaArray, которые представляют массивы Java, состоящие из целых значений. (Чтобы узнать имена и типы этих свойств, посмотрите документацию по *java.awt.Polygon* в справочном руководстве по Java.) Мы можем использовать эти объекты JavaArray для случайной инициализации объекта Polygon с помощью следующего кода:

```
for(var i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random()*100);
for(var i = 0; i < p.ypoints.length; i++)
    p.ypoints[i] = Math.round(Math.random()*100);
```

22.4.5. Методы Java

Классы JavaClass и JavaObject позволяют нам вызывать соответственно статические методы и методы экземпляра. В Netscape 3 для внутреннего представления методов Java применялся объект JavaMethod. Однако в Netscape 4 методы Java стали просто обычными методами, как методы встроенных объектов JavaScript, таких как String и Date.

Работая с методами Java, помните, что им требуется фиксированное количество аргументов фиксированных типов. Если количество или тип переданных аргументов неправильно, вы получите ошибку JavaScript.

22.5. Преобразование данных в LiveConnect

Java – строго типизированный язык с относительно большим количеством типов данных, а JavaScript – нетипизированный язык с относительно небольшим количеством типов. Вот почему одной из главных обязанностей LiveConnect является преобразование данных. Когда JavaScript устанавливает значения поля класса Java или экземпляра или передает аргумент в метод Java, значение JavaScript должно быть преобразовано в эквивалентное значение Java, а когда JavaScript читает поле класса Java или экземпляра или получает значение, возвращаемое методом Java, это значение Java должно быть преобразовано в совместимое значение JavaScript.[1]

---

[1] Кроме того, преобразование данных может произойти, когда Java читает или устанавливает значение поля JavaScript или вызывает метод JavaScript. Эти преобразования, однако, выполняются по-другому и описаны в этой главе позднее – там же, где и вызов JavaScript из Java. Сейчас мы обсуждаем только преобразование данных, выполняемое, когда код JavaScript взаимодействует с Java, а не наоборот.

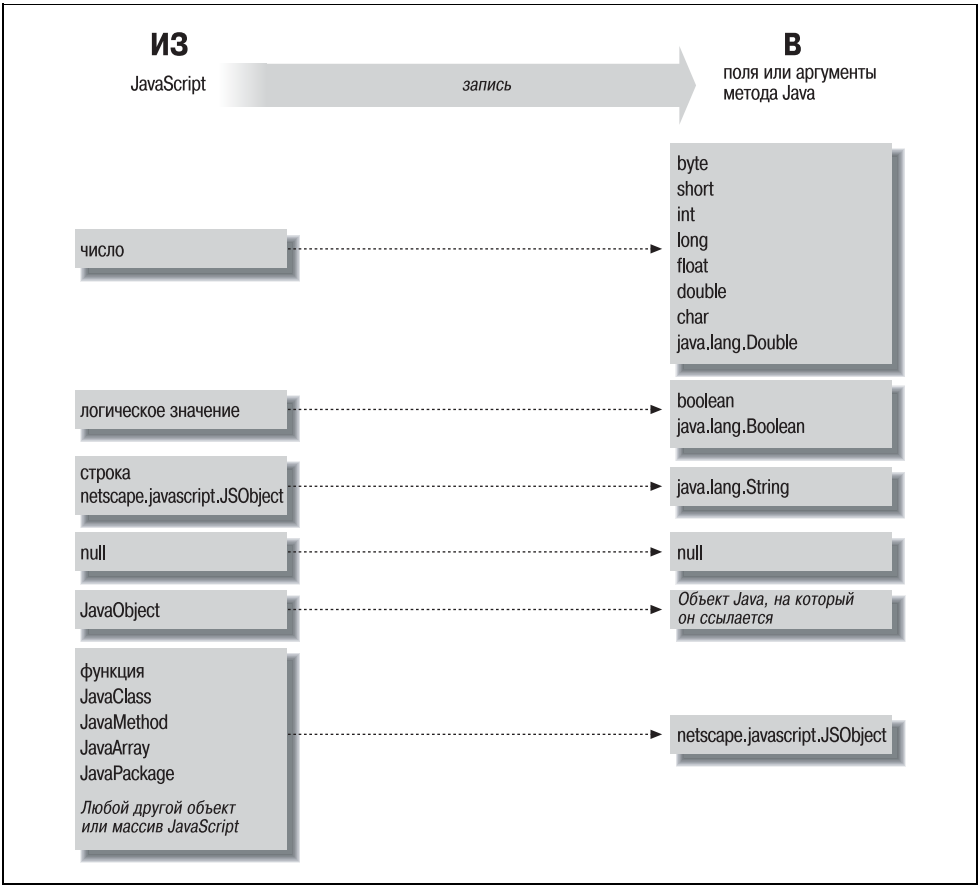

Рис. 22.2. *Преобразование данных, выполняемое JavaScript при записи значений Java*

Преобразование данных, выполняемое при записи и чтении значений Java из JavaScript, показано соответственно на рис. 22.2 и 22.3.

Обратите внимание на следующие моменты в преобразовании данных, иллюстрируемые рис. 22.2:

- На рис. 22.2 не показаны все возможные преобразования между типами JavaScript и Java. Дело в том, что перед преобразованием из JavaScript в Java могут происходить внутренние преобразования JavaScript. Например, если вы передаете число JavaScript методу Java, ожидающему аргумент типа *java.lang.String*, JavaScript сначала преобразует число в строку JavaScript, которая затем может быть преобразована в строку Java.
- Число JavaScript может быть преобразовано в любой элементарный числовой тип Java. Фактическое выполняемое преобразование зависит, конечно, от типа поля Java или аргумента метода, которому передается значение.

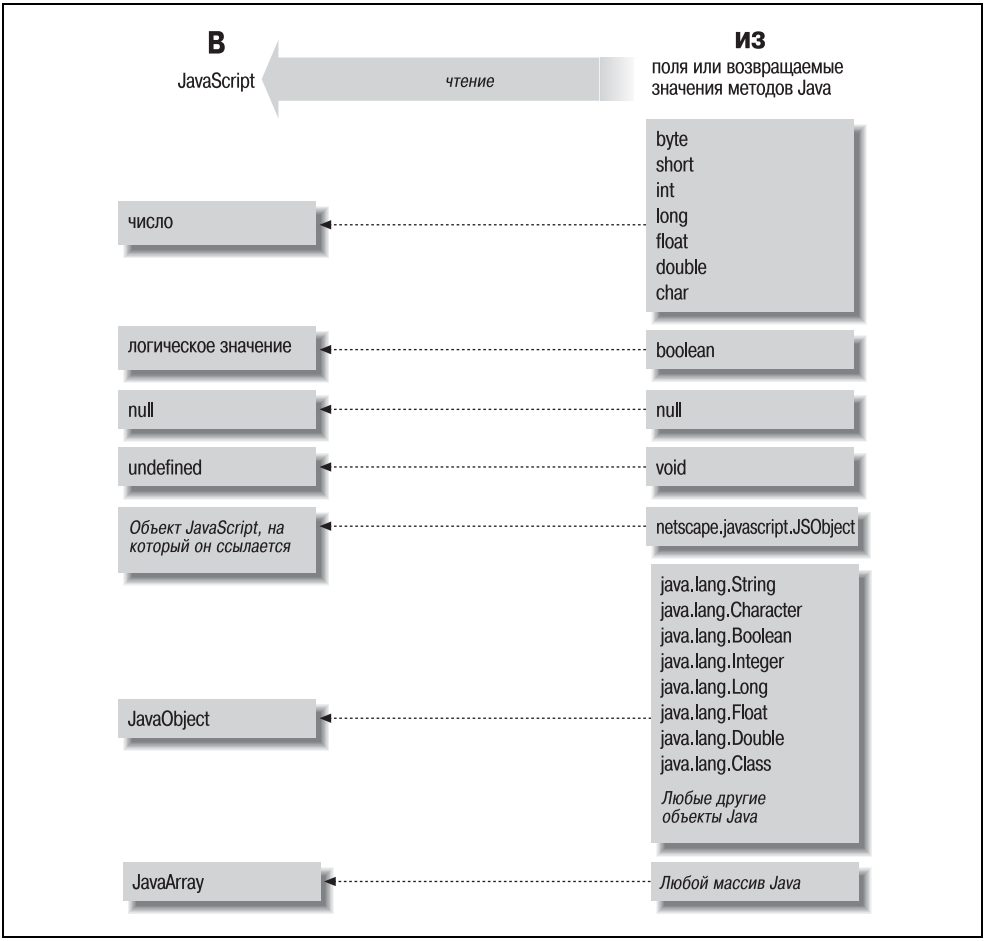

Рис. 22.3. *Преобразование данных, выполняемое JavaScript при чтении значений Java*

Обратите внимание, что при этом может теряться точность, если, например, передается большое число полю Java с типом `short` или значение с плавающей точкой целочисленному типу Java.

- Число JavaScript может быть также преобразовано в экземпляр класса Java *java.lang.Double*, но не в экземпляр другого схожего класса, такого как *java.lang.Integer* или *java.lang.Float*.
- В JavaScript нет представления для символьных данных, поэтому число JavaScript может также быть преобразовано в элементарный Java-тип `char`.
- JavaObject в JavaScript при передаче в Java освобождается от «обертки», то есть преобразуется в объект Java, который он представляет. Однако заметьте, что объекты JavaClass в JavaScript не преобразуются в экземпляры *java.lang.Class*, как можно было ожидать.

- Массивы JavaScript не преобразуются в массивы Java.

Также отметим следующие моменты в отношении преобразований, изображенных на рис. 22.3:

- Так как JavaScript не имеет типа для символьных данных, элементарный тип Java `char` преобразуется в число JavaScript, а не в строку, как можно было бы ожидать.
- Экземпляр *java.lang.Double*, *java.lang.Integer* или другого аналогичного класса Java не преобразуется в число JavaScript. Как любой объект Java, он преобразуется в JavaScript в объект JavaObject.
- Строка Java является экземпляром *java.lang.String*, поэтому, как любой другой объект Java, преобразуется в JavaObject, а не в реальную строку JavaScript.
- Любой массив Java преобразуется в JavaScript в объект JavaArray.

22.5.1. Объекты-обертки

Еще одно понятие, связанное с рис. 22.2 и 22.3, – это объекты-обертки. Если преобразования между большинством элементарных типов JavaScript и Java возможны, то преобразования между объектными типами – как правило, нет. Поэтому в LiveConnect определяется для JavaScript объект JavaObject – он представляет объект Java, который не может быть непосредственно преобразован в объект JavaScript. В каком-то смысле JavaObject – это обертка JavaScript вокруг объекта Java. Когда JavaScript читает данные Java (поле или возвращаемое методом значение), любые объекты Java упаковываются, и JavaScript видит объект JavaObject.

То же самое происходит, когда объект JavaScript записывается в поле Java или передается в метод Java. Нет способа преобразовать объект JavaScript в объект Java, поэтому объект упаковывается. Java-«обертка» для объекта JavaScript – это класс Java *netscape.javascript.JSObject*.

Становится еще интереснее, когда объекты-обертки передаются обратно. Если JavaScript записывает JavaObject в поле Java или передает его методу Java, LiveConnect сначала распаковывает объект, преобразуя JavaObject обратно в объект Java, который он представляет. Аналогично, если JavaScript читает поле Java или получает значение, возвращаемое методом Java, являющееся экземпляром *netscape.javascript.JSObject*, этот JSObject также распаковывается для извлечения и возвращения исходного объекта JavaScript.

22.5.2. Преобразование данных LiveConnect в Netscape 3

В Netscape 3 имеется ошибка в способе, которым LiveConnect преобразует значения Java в значения JavaScript: значение элементарного поля объекта Java возвращается в виде объекта JavaScript, а не в виде элементарного значения JavaScript. Например, если JavaScript читает значение поля типа `int`,

LiveConnect в Netscape 3 преобразует это значение в объект Number, а не в элементарное числовое значение. Аналогично, LiveConnect преобразует значение полей Java с типом `boolean` в объекты JavaScript Boolean, а не в элементарные логические значения. Обратите внимание, что эта ошибка возникает только при чтении данных из полей Java. Она не возникает, когда LiveConnect преобразует значение, возвращаемое методом Java.

Объекты Number и Boolean ведут себя почти (но не точно) как элементарные числовые и логические значения. Отличие в том, что объекты Number, как все объекты JavaScript, используют оператор + для конкатенации строк, а не для сложения. И выполнение кода, подобного приведенному ниже, использующего LiveConnect в Netscape 3, может дать неожиданный результат:

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width;    // Это объект Number, а не элементарное числовое значение.
var new_w = w + 1;  // Ой! new_w теперь равно "51", а не 6, как ожидалось.
```

Чтобы обойти эту проблему, вы можете явно вызвать метод `valueOf()` для преобразования объекта Number в соответствующее числовое значение. Например:

```
var r = new java.awt.Rectangle(0,0,5,5);
var w = r.width.valueOf();  // Теперь получилось элементарное числовое значение.
var new_w = w + 1;          // Теперь new_w равно 6, как и требовалось.
```

22.6. Преобразование JavaObject в JavaScript

Разобравшись с непростым предыдущим разделом, вы можете ожидать, что с темой преобразования данных покончено. К сожалению, нам следует еще обсудить, как JavaScript преобразует объекты JavaObject в различные элементарные типы JavaScript. На рис. 22.3 можно заметить, что многие типы данных Java, включая строки Java (экземпляры *java.lang.String*), преобразуются в JavaScript в объекты JavaObject, а не в элементарные типы данных JavaScript, такие как строки. Это значит, что при использовании LiveConnect вам часто придется работать с объектами JavaObject.

Обратимся снова к табл. 11.1, в которой показано, как преобразуются различные типы данных JavaScript при использовании в различных контекстах. Например, когда число используется в строковом контексте, оно преобразуется в строку, а когда объект используется в логическом контексте, он преобразуется в значение `false`, если он равен `null`, и в `true` в противном случае. Эти правила преобразования не применяются к объектам JavaObject, которые преобразуются по своим собственным правилам, приведенным ниже:

- Когда JavaObject используется в числовом контексте, он преобразуется в число путем вызова метода `doubleValue()` представляемого им объекта Java. Если в объекте Java этот метод не определен, возникает ошибка JavaScript.
- В логическом контексте объект JavaObject преобразуется в логическое значение при помощи метода `booleanValue()` представляемого им объекта

Java. Если в объекте Java этот метод не определен, возникает ошибка JavaScript.

- В строковом контексте объект JavaObject преобразуется в строковое значение путем вызова метода `toString()` представляемого им объекта Java. Все объекты Java определяют или наследуют этот метод, поэтому такое преобразование всегда оказывается успешным.
- В объектном контексте для объекта JavaObject никакое преобразование не требуется, так как он уже является объектом JavaScript.

Из-за этих специальных правил преобразования, а также по другим причинам объекты JavaObject ведут себя не так, как другие объекты JavaScript, и есть несколько подводных камней, о которых вы должны знать. Во-первых, не принято работать с объектами JavaObject, представляющими экземпляры *java.lang.Double* или другие числовые объекты. Во многих отношениях такой объект JavaObject ведет себя так же, как элементарное числовое значение, но будьте аккуратны при использовании оператора +. Когда вы используете объект JavaObject (или любой объект JavaScript) с оператором +, вы задаете строковый контекст, поэтому объект преобразуется в строку для выполнения конкатенации строк, а не в число для выполнения сложения.

Когда необходимо явно преобразовать объект JavaScript в элементарное значение, обычно вызывается его метод `valueOf()`. Заметьте, что для объектов JavaObject это не годится. Мы уже говорили, что в классе JavaObject не определены собственные свойства, все его свойства представляют поля и методы представляемого им объекта Java. То есть объекты JavaObject не поддерживают обычные методы JavaScript, такие как `valueOf()`. В случае нашего объекта *java.lang.Double* в обертке JavaObject, когда вам требуется привести объект к элементарному значению, вы должны вызвать метод Java `doubleValue()`.

Еще одно различие между объектами JavaObject и другими типами данных JavaScript состоит в том, что JavaObject могут выступать в логическом контексте, только если в них определен метод `booleanValue()`. Предположим, что `button` – это переменная JavaScript, которая содержит либо `null`, либо объект JavaObject, представляющий экземпляр класса *java.awt.Button*. Чтобы проверить, содержит ли переменная `null`, можно по привычке написать:

```
if (!button) { ... }
```

Пока значение `button` равно `null`, все прекрасно. Но если `button` действительно содержит JavaObject, представляющий экземпляр *java.awt.Button*, LiveConnect попытается вызвать метод `booleanValue()`. Когда выяснится, что в *java.awt.Button* он не определен, возникает ошибка JavaScript. Решение в этом случае состоит в явном указании цели вашей проверки, чтобы избежать использования JavaObject в логическом контексте:

```
if (button != null) { ... }
```

Это в любом случае полезная привычка, так как в результате читать и понимать код становится легче.

22.7. Преобразование данных из Java в JavaScript

В последних двух разделах рассмотрены правила, по которым значения преобразуются при чтении и записи значений полей Java и вызове методов Java из JavaScript. Эти правила определяют, как объекты JavaObject, JavaArray и JavaClass преобразуют данные; они применяются только к ситуации, когда JavaScript манипулирует данными Java. Когда Java манипулирует данными JavaScript, преобразование выполняется классом Java JSObject, и правила преобразования другие. Эти преобразования показаны на рис. 22.4 и рис. 22.5.

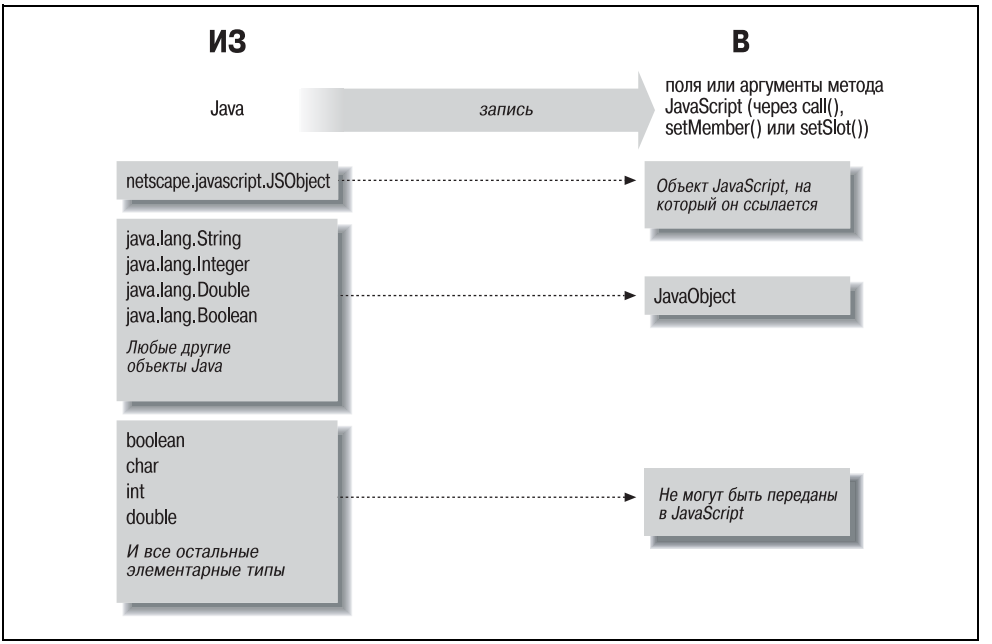

Рис. 22.4. *Преобразования данных, выполняемые при записи значений JavaScript из кода Java*

Изучая эти рисунки, следует помнить, что Java может взаимодействовать с JavaScript только через API, предоставляемый классом JSObject. Так как Java – строго типизированный язык, методы, определенные в этом классе, могут работать только с объектами Java, а не со значениями элементарных типов. Например, когда вы читаете значение числа JavaScript, метод `getMember()` возвращает объект `java.lang.Double`, а не элементарное значение `double`.

При написании функций JavaScript, которые вызываются из Java, имейте в виду, что аргументы, переданные из Java, – это либо объекты JavaScript, полученные из обертки Java – JSObject, либо JavaObject. LiveConnect просто не

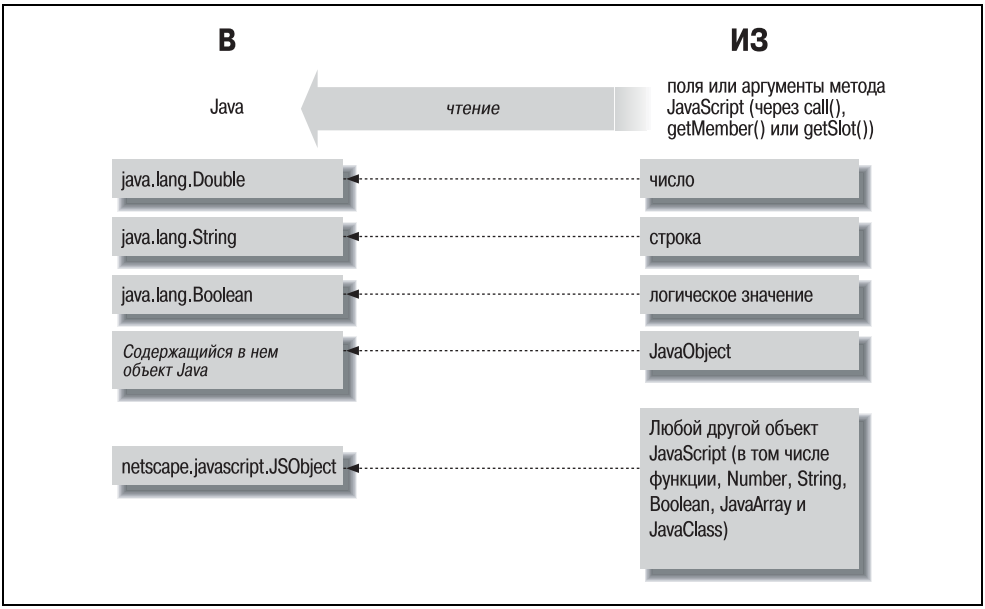

Рис. 22.5. *Преобразования данных, выполняемые при чтении значений JavaScript из кода Java*

разрешает Java передавать в качестве аргументов метода элементарные значения. Мы уже знаем из этой главы, что объекты JavaObject ведут себя не совсем так, как другие объекты. Например, экземпляр *java.lang.Double* ведет себя не так, как элементарное числовое значение JavaScript или даже объект JavaScript Number. При работе со свойствами JavaScript, значения которых установлены в Java, необходимо соблюдать определенные предосторожности.

Один из способов избежать всех трудностей с преобразованием данных состоит в том, чтобы использовать для взаимодействия кода Java с JavaScript метод `eval()` класса JSObject. Для этого код Java должен преобразовать все аргументы методов и значения свойств в строковый формат. Затем строка для исполнения может быть без изменений передана в JavaScript, где строковый формат данных преобразуется в нужные значения JavaScript.

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-055-3, название «JavaScript. Подробное руководство, 4-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.

Часть III. Справочник по базовому JavaScript

Эта часть книги представляет собой полный справочник по всем объектам, свойствам, функциям, методам и обработчикам событий базового языка JavaScript. На первых страницах рассказывается, как работать с этим справочным материалом.

Справочник по базовому JavaScript

Здесь описаны классы, методы и свойства, определенные в базовом языке JavaScript. Вводная часть и образец справочной статьи призваны помочь вам понять, как работать со справочником и извлечь из него максимум. Не поленитесь и внимательно прочитайте этот материал – вам будет легче искать и использовать нужную информацию!

Материал справочника организован в алфавитном порядке. Справочные статьи, посвященные методам и свойствам классов, отсортированы по их полным именам, включающим имена определяющих их классов. Так, для того чтобы найти метод `replace()` класса String, следует искать «String.replace», а не «replace».

В базовом JavaScript определены некоторые глобальные функции и свойства, такие как `eval()` и `NaN`. Формально они представляют собой свойства глобального объекта. Однако у глобального объекта нет имени, поэтому в справочнике они перечислены по их неуточненным именам. Для удобства полный набор глобальных функций и свойств базового JavaScript объединен в специальную справочную статью «Global» (хотя объекта или класса с таким именем нет).

Может оказаться, что вы не помните имени класса или интерфейса, определяющего метод или свойство, которое ищете, или не знаете точно, в какой справочной статье искать данный класс или интерфейс. Часть VI этой книги представляет собой специальный указатель, предназначенный для оказания помощи в таких ситуациях. Найдите имя класса, метода или свойства, и там будет сказано, какой класс и в какой справочной статье надо искать. Допустим, надо найти «Date» – в части VI сказано, что класс Date описан в справочнике по базовому JavaScript («базовый»). А если вы ищете имя «match», то в части VI вы узнаете, что `match()` – это метод класса String, и он также описан в этом справочнике.

Найдя искомую справочную статью, вы без особых трудностей найдете и нужную информацию. Но вам будет легче работать с этим справочником, если вы поймете, как написаны и организованы справочные статьи. Далее, после заголовка «Пример справочной статьи», показана структура всех справочных статей и рассказано, какую информацию можно в этих статьях найти. Прежде чем искать что-то в справочнике, не пожалейте времени на ознакомление с этим примером.

Пример справочной статьи

Доступность

как читать справочные статьи по базовому JavaScript **наследуется от/перекрывает**

Заголовок и краткое описание

Каждая статья справочника начинается с приведенного выше блока заголовка, состоящего из четырех частей. Статьи отсортированы по заголовку. Краткое описание под заголовком дает общее представление о предмете, описанном в данной статье, и позволяет быстро понять, интересует ли вас ее оставшаяся часть.

Доступность

Информация о доступности приведена в правом верхнем углу блока заголовка и говорит о том, в какой версии интерпретаторов Netscape JavaScript и Microsoft JScript был реализован предмет статьи (класс, метод или свойство). Если он стандартизован в ECMAScript, то будет сказано, в какой версии стандарта он введен. Можно считать, что все, что доступно в одной версии JavaScript, доступно и в более поздних его версиях. Однако имейте в виду, что если здесь указано, что предмет статьи признан устаревшим, то он может быть удален в будущем, и применять его не следует.

Наследует/перекрывает

Если класс является наследником суперкласса или перекрывает метод суперкласса, эта информация отображается в правом нижнем углу блока заголовка.

В главе 8 говорилось, что классы JavaScript могут наследовать свойства и методы от других классов. Например, класс String является наследником Object, а класс RangeError – наследником Error, а он в свою очередь – наследником Object. Благодаря этой информации понятно, что, возможно, надо просмотреть разделы, посвященные перечисленным суперклассам.

Если метод имеет то же имя, что и метод суперкласса, он перекрывает его. В качестве примера см. статью «Array.toString()».

Конструктор

Если страница справочника описывает класс, то она, как правило, содержит раздел «Конструктор», в котором показано, как метод-конструктор применяется для создания экземпляров класса. Так как конструкторы являются разновидностью методов, раздел «Конструктор» во многом похож на раздел «Синтаксис» в справочной статье, посвященной методу.

Синтаксис

В справочных статьях функций, методов и свойств есть раздел «Синтаксис», показывающий, как использовать эту функцию, метод или свойство в программе. Вот, например, синтаксис метода `Array.concat()`:

```
массив.concat(значение, ...)
```

Курсивом набран текст, который должен быть заменен чем-то другим. *массив* должен быть заменен переменной, в которой содержится массив, или выражением JavaScript, результатом которого является массив. А *значение* просто представляет собой произвольное значение, которое будет добавлено в массив. Многоточие (...) указывает, что данный метод может принимать любое число аргументов *значение*. Так как термы `concat`, открывающая и закрывающая скобки набраны не *курсивом*, они должны включаться в код JavaScript точно так, как показано здесь.

Аргументы

Если в справочной статье описывается функция, метод или класс с методом-конструктором, то за разделами «Конструктор» и «Синтаксис» следует подраздел «Аргументы», в котором описаны аргументы метода, функции или конструктора. Если аргументы отсутствуют, этот подраздел опускается.

аргумент1

Здесь приводится описание перечисленных аргументов. Это, например, описание аргумента *аргумент1*.

аргумент2

Это описание аргумента *аргумент2*.

Возвращаемое значение

Если конструктор, функция или метод возвращает какое-либо значение, в данном подразделе приведено описание этого значения.

Исключения

Если конструктор, функция или метод может генерировать исключения, в этом подразделе перечислены типы возможных исключений и описаны обстоятельства, при которых они могут возникать.

Свойства

Если справочная страница описывает класс, в разделе «Свойства» перечислены свойства, определенные в этом классе, и приведено краткое описание каждого из них. В этом справочнике по базовому JavaScript для каждого свойства также имеется собственная полная справочная статья. Например, в справочной статье по классу Array в данном разделе указано свойство `length` и дается его краткое описание, но полное описание этого свойства находится в справочной статье «Array.length». Список свойств выглядит следующим образом:

`свойство1`

Это краткое описание `свойства1`, включающее его тип, смысл или назначение независимо от того, доступно ли оно только для чтения или для чтения и записи.

`свойство2`

То же самое о свойстве `свойство2`.

Методы

Справочная страница класса, определяющего методы, включает раздел «Методы». Он во многом похож на раздел «Свойства» за исключением того, что в нем перечислены не свойства, а методы. Для всех методов также имеются отдельные справочные страницы.

Описание

Большинство справочных страниц также содержат раздел «Описание», являющийся основным описанием класса, метода, функции или свойства, которому посвящена страница. Это основная часть справочной страницы. Те, кто до сих пор ничего не знал о классе, методе или свойстве, могут сразу перейти к этому разделу, а затем вернуться и посмотреть предыдущие разделы, такие как «Аргументы», «Свойства» и «Методы». Тем же, кто знаком с классом, методом или свойством, возможно, читать этот раздел не надо, поскольку им требуется лишь быстро найти определенную информацию о нем (например, из разделов «Аргументы» или «Свойства»).

В некоторых статьях этот раздел представляет собой лишь короткий абзац. В других он может занимать одну или несколько страниц. Для некоторых простых методов разделы «Аргументы» и «Возвращаемое значение» сами достаточно хорошо описывают метод, и тогда раздел «Описание» опускается.

Пример

Некоторые справочные страницы включают пример, иллюстрирующий типичное применение. Большинство страниц, однако, не содержат примеров. Вы найдете их в первой половине этой книги.

Ошибки

Если предмет справочной статьи работает не совсем правильно, то в данном разделе приведено описание ошибок. Однако заметим, что в данной книге не ставилась задача учесть все ошибки во всех версиях и реализациях JavaScript.

Смотри также

Многие справочные страницы заканчиваются перекрестными ссылками на связанные справочные страницы, которые могут представлять интерес. Иногда справочные страницы ссылаются на одну из глав книги.

arguments[]

JavaScript 1.1; JScript 2.0; ECMAScript v1

массив аргументов функции

Синтаксис

`arguments`

Описание

Массив `arguments[]` определен только внутри тела функции, где он ссылается на объект Arguments этой функции. Данный объект имеет нумерованные свойства и служит в качестве массива, содержащего все переданные функции аргументы. Идентификатор `arguments` – это, по существу, локальная переменная, автоматически объявляемая и инициализируемая внутри каждой функции. Она ссылается на объект Arguments только внутри тела функции и не определена в глобальном коде.

Смотри также

Arguments; главу 7

Arguments

JavaScript 1.1; JScript 2.0; ECMAScript v1

аргументы и другие свойства функции **наследуется от Object**

Синтаксис

`arguments`

`arguments[n]`

Элементы

Объект Arguments определен только внутри тела функции. Хотя формально он не является массивом, у него есть нумерованные свойства, работающие как элементы массива, и свойство `length`, равное количеству элементов массива. Его элементами явля-

ются значения, переданные функции в качестве аргументов. Элемент 0 – это первый аргумент, элемент 1 – второй аргумент и т. д. Все значения, переданные в качестве аргументов, становятся элементами массива в объекте Arguments, независимо от того, присвоены ли этим аргументам имена в объявлении функции.

Свойства

`callee`

Ссылка на исполняемую в данный момент функцию.

`length`

Количество аргументов, переданных функции и количество элементов массива в объекте Arguments.

Описание

Когда вызывается функция, для нее создается объект Arguments, и локальная переменная `arguments` автоматически инициализируется ссылкой на объект Arguments, что дает возможность определить, сколько аргументов было передано функции, и обратиться к неименованным аргументам. Свойство `callee` позволяет неименованной функции ссылаться на саму себя.

Для большинства задач объект Arguments можно рассматривать как массив с дополнительным свойством `callee`. Однако он не является экземпляром объекта Array, а свойство `Arguments.length` не ведет себя особым образом, как свойство `Array.length`, и не может использоваться для изменения размера массива.

Объект Arguments имеет одну *очень* необычную особенность. Когда у функции есть именованные аргументы, элементы массива в объекте Arguments являются синонимами локальных переменных, содержащих аргументы функции. Объект Arguments и имена аргументов предоставляют два различных способа обращения к одной и той же переменной. Изменение значения аргумента с помощью имени аргумента изменяет значение, извлекаемое через объект Arguments, а изменение аргумента через объект Arguments изменяет значение, извлекаемое по имени аргумента.

Смотри также

Function; главу 7

Arguments.callee

JavaScript 1.2; JScript 5.5; ECMAScript v1

работающая в данный момент функция

Синтаксис

`arguments.callee`

Описание

`arguments.callee` ссылается на функцию, работающую в данный момент. Она предоставляет способ ссылки на себя для неименованной функции. Это свойство определено только внутри тела функции.

Пример

```
// Неименованный функциональный литерал использует свойство callee для
// ссылки на себя, чтобы он мог быть рекурсивным
var factorial = function(x) {
```

```
    if (x < 2) return 1;
    else return x * arguments.callee(x-1);
}
var y = factorial(5);  // Возвращает 120
```

Arguments.length

JavaScript 1.1; JScript 2; ECMAScript v1

количество переданных функции аргументов

Синтаксис

`arguments.length`

Описание

Свойство `length` объекта Arguments возвращает количество аргументов, переданных текущей функции. Это свойство определено только внутри тела функции.

Заметьте, что это свойство возвращает фактическое количество переданных аргументов, а не ожидаемое. О количестве аргументов в объявлении функции говорится в статье «Function.length». Заметьте также, что это свойство не ведет себя особым образом, как свойство `Array.length`.

Пример

```
// Используем объект Arguments, чтобы проверить, верное ли
// количество аргументов было передано
function check(args) {
    var actual = args.length;           // Фактическое количество аргументов
    var expected = args.callee.length;  // Ожидаемое количество аргументов
    if (actual != expected) {           // Если они не совпадают, возбуждаем
                                        // исключительную ситуацию
        throw new Error("Неверное число аргументов: ожидается: " +
                        expected + "; фактически передано " + actual);
    }
}
// Функция, демонстрирующая использование функции, приведенной выше
function f(x, y, z) {
    check(arguments);  // Проверяем правильность количества переданных аргументов
    return x + y + z;  // Теперь выполняем оставшуюся часть функции
                       // обычным образом
}
```

Смотри также

Array.length, Function.length

Array

JavaScript 1.1; JScript 2.0; ECMAScript v1

встроенная поддержка массивов **наследуется от Object**

Конструктор

`new Array()`

`new Array(`*размер*`)`

`new Array(`*элемент0*`, `*элемент1*`, ..., `*элементn*`)`

Аргументы

размер

Желаемое количество элементов в массиве. Длина возвращаемого массива (`length`) равна аргументу *размер*.

элемент0, ... элементn

Список аргументов из двух и более произвольных значений. Когда `Array()` вызывается с этими аргументами, элементы только что созданного массива инициализируются указанными значениями, а свойство `length` становится равным количеству аргументов.

Возвращаемое значение

Только что созданный и проинициализированный массив. Когда `Array()` вызывается без аргументов, возвращаемый массив пуст и значение свойства `length` равно 0. При вызове с одним числовым аргументом конструктор возвращает массив с указанным количеством неопределенных элементов. При вызове с любыми другими аргументами конструктор инициализирует массив значениями, указанными его аргументами. Когда конструктор `Array()` вызывается как функция, без оператора `new`, он ведет себя точно так же, как при вызове с оператором `new`.

Исключения

`RangeError`

Когда конструктору `Array()` передается один целый аргумент *размер*, то генерируется исключение `RangeError`, если *размер* отрицателен или превышает $2^{32}-1$.

Синтаксис литерала

ECMAScript v3 определяет, а JavaScript 1.2 и JScript 3.0 реализуют синтаксис литералов для массивов. Программист может создавать и инициализировать массив, заключая список выражений, перечисленных через запятые, в квадратные скобки. Значения этих выражений становятся элементами массива. Например:

```
var a = [1, true, 'abc'];
var b = [a[0], a[0]*2, f(x)];
```

Свойства

`length`

Целое, доступное для чтения и для записи, задает количество элементов массива, или, если элементы массива расположены не непрерывно, число, на единицу большее индекса последнего элемента массива. Изменение этого свойства укорачивает или расширяет массив.

Методы

`concat()`

Присоединяет элементы к массиву.

`join()`

Преобразует все элемента массива в строки и выполняет их конкатенацию.

`pop()`

Удаляет элемент из конца массива.

`push()`

Помещает элемент в конец массива.

`reverse()`

Меняет порядок элементов массива на противоположный.

`shift()`

Сдвигает элементы к началу массива.

`slice()`

Возвращает подмассив массива.

`sort()`

Сортирует элементы массива.

`splice()`

Вставляет, удаляет или заменяет элементы массива.

`toLocaleString()`

Преобразует массив в локализованную строку.

`toString()`

Преобразует массив в строку.

`unshift()`

Вставляет элементы в начало массива.

Описание

Массивы – это базовое средство JavaScript, подробно описанное в главе 9.

Смотри также

Главу 9

Array.concat()

JavaScript 1.2; JScript 3.0; ECMAScript v3

выполняет конкатенацию массивов

Синтаксис

массив`.concat(`*значение*`, ...)`

Аргументы

значение, ...

Любое количество значений, присоединяемых к *массиву*.

Возвращаемое значение

Новый массив, образуемый присоединением к *массиву* каждого из указанных аргументов.

Описание

`concat()` создает и возвращает новый массив, являющийся результатом присоединения каждого из его аргументов к *массиву*. Этот метод не изменяет *массив*. Если какие-либо из аргументов `concat()` сами являются массивами, то присоединяются элементы этих массивов, а не сами массивы.

Пример

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

Смотри также

Array.join(), Array.push(), Array.splice()

Array.join()

JavaScript 1.1; JScript 2.0; ECMAScript v1

выполняет конкатенацию элементов массива в строку

Синтаксис

массив.join()

массив.join(*разделитель*)

Аргументы

разделитель

Необязательный символ или строка, выступающая в качестве разделителя элементов в результирующей строке. Если аргумент опущен, используется запятая.

Возвращаемое значение

Строка, получающаяся в результате преобразования каждого элемента *массива* в строку и объединения их с *разделителем* между элементами путем конкатенации.

Описание

join() преобразует каждый из элементов массива в строку и затем выполняет конкатенацию этих строк, вставляя указанную строку-*разделитель* между элементами. Метод возвращает результирующую строку.

Обратное преобразование (разбиение строки на элементы массива) можно выполнить посредством метода split() объекта String. Подробности см. в справочной статье по методу String.split().

Пример

```
a = new Array(1, 2, 3, "testing");
s = a.join("+");  // s - это строка "1+2+3+testing"
```

Смотри также

String.split()

Array.length

JavaScript 1.1, JScript 2.0; ECMAScript v1

размер массива

Синтаксис

массив.length

Описание

Свойство length массива всегда на единицу больше индекса последнего элемента, определенного в массиве. Для традиционных «плотных» массивов, имеющих последовательные элементы и начинающихся с элемента 0, length указывает количество элементов в массиве.

Свойство length инициализируется в момент создания массива с помощью метода-конструктора Array(). Добавление новых элементов изменяет значение length:

```
a = new Array();                      // a.length равно 0
b = new Array(10);                    // b.length равно 10
c = new Array("one", "two", "three"); // c.length равно 3
c[3] = "four";                        // c.length изменяется на 4
c[10] = "blastoff";                   // c.length становится равным 11
```

Можно установить значение свойства length для изменения размера массива. Если новое значение length меньше предыдущего, массив обрезается и элементы в его конце теряются. Если значение length увеличивается (новое значение больше старого), массив становится больше, а новые элементы, добавленные в конец массива, получают значение undefined.

Array.pop()

JavaScript 1.2; JScript 5.5; ECMAScript v3

удаляет и возвращает последний элемент массива

Синтаксис

массив.pop()

Возвращаемое значение

Последний элемент *массива*.

Описание

pop() удаляет последний элемент *массива*, уменьшает длину массива на единицу и возвращает значение удаленного элемента. Если массив уже пуст, pop() его не изменяет и возвращает значение undefined.

Пример

Метод pop() и его компаньон push() позволяют реализовать стек, работающий по принципу «первым вошел, последним вышел». Например:

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

Смотри также

Array.push()

Array.push()

JavaScript 1.2; JScript 5.5; ECMAScript v3

добавляет элементы к массиву

Синтаксис

массив.push(*значение*, ...)

Аргументы

значение, ...

Одно или более значений, которые должны быть добавлены в конец *массива*.

Возвращаемое значение

Новая длина массива, после добавления в него указанных значений.

Описание

Метод push() добавляет свои аргументы в указанном порядке в конец *массива*. Он изменяет существующий *массив*, а не создает новый. push() и его метод-компаньон pop() используют массив для реализации стека, работающего по принципу «первым вошел, последним вышел». Пример см. в статье «Array.pop()».

Ошибки

В реализациях JavaScript от Netscape, когда явно запрашивается версия языка 1.2, эта функция возвращает последнее добавленное значение, а не новую длину массива.

Смотри также

Array.pop()

Array.reverse()

JavaScript 1.1; JScript 2.0; ECMAScript v1

меняет порядок элементов в массиве на противоположный

Синтаксис

массив.reverse()

Описание

Метод reverse() объекта Array меняет порядок элементов в массиве на противоположный. Он делает это «на месте», то есть переупорядочивает элементы указанного *массива*, не создавая новый. Если есть несколько ссылок на *массив*, новый порядок элементов массива будет виден через все ссылки.

Пример

```
a = new Array(1, 2, 3);  // a[0] == 1, a[2] == 3;
a.reverse();             // Теперь a[0] == 3, a[2] == 1;
```

Array.shift()

JavaScript 1.2; JScript 5.5; ECMAScript v3

сдвигает элементы к началу массива

Синтаксис

массив.shift()

Возвращаемое значение

Бывший первый элемент массива.

Описание

`shift()` удаляет и возвращает первый элемент *массива*, смещая все последующие элементы на одну позицию вниз для занятия освободившегося места в начале массива. Если массив пуст, `shift()` не делает ничего и возвращает значение `undefined`. Обратите внимание, что `shift()` не создает новый массив, а непосредственно изменяет сам *массив*.

`shift()` похож на `Array.pop()`, за исключением того, что работает с началом массива, а не с его концом. `shift()` часто применяется в сочетании с `unshift()`.

Пример

```
var a = [1, [2,3], 4]
a.shift();  // Возвращает 1; a = [[2,3], 4]
a.shift();  // Возвращает [2,3]; a = [4]
```

Смотри также

Array.pop(), Array.unshift()

Array.slice() JavaScript 1.2; JScript 3.0; ECMAScript v3

возвращает фрагмент массива

Синтаксис

массив`.slice(`*начало*`, `*конец*`)`

Аргументы

начало

Индекс элемента массива, с которого начинается фрагмент. Отрицательное значение этого аргумента указывает позицию, измеряемую от конца массива. Другими словами, –1 обозначает последний элемент, –2 – второй элемент с конца и т. д.

конец

Индекс элемента массива, расположенного непосредственно после конца фрагмента. Если этот аргумент не указан, фрагмент включает все элементы массива от элемента, заданного аргументом *начало*, до конца массива. Если этот аргумент отрицателен, позиция элемента отсчитывается от конца массива.

Возвращаемое значение

Новый массив, содержащий элементы *массива* от элемента, заданного аргументом *начало*, до элемента, определяемого аргументом *конец*, но не включая его.

Описание

`slice()` возвращает фрагмент, или подмассив, *массива*. Возвращаемый массив содержит элемент, заданный аргументом *начало*, и все последующие элементы до элемента, заданного аргументом *конец*, но не включая его. Если аргумент *конец* не указан, возвращаемый массив содержит все элементы от элемента, заданного аргументом *начало*, до конца *массива*.

Обратите внимание, что `slice()` не изменяет массив. Для удаления фрагмента массива применяется `Array.splice()`.

Пример

```
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]; в IE 4 работает с ошибкой: возвращает [1,2,3]
```

Ошибки

В Internet Explorer 4 *начало* не может быть отрицательным числом.

Смотри также

Array.splice()

Array.sort()

JavaScript 1.1; JScript 2.0; ECMAScript v1

сортирует элементы массива

Синтаксис

массив.sort()

массив.sort(*orderfunc*)

Аргументы

orderfunc

Необязательная функция, задающая порядок сортировки.

Возвращаемое значение

Ссылка на массив. Обратите внимание, что массив сортируется на месте, и копия не делается.

Описание

Метод sort() сортирует элементы *массива* на месте – копия массива не делается. Если sort() вызывается без аргументов, элементы массива располагаются в алфавитном порядке (точнее, в порядке, определяемом используемой в системе кодировкой символов). Если необходимо, элементы сначала преобразуются в строки, чтобы их можно было сравнивать.

Для того чтобы отсортировать элементы массива в каком-либо другом порядке, необходимо указать функцию сравнения, которая сравнивает два значения и возвращает число, обозначающее их относительный порядок. Функция сравнения должна принимать два аргумента, *a* и *b*, и вернуть одно из следующих значений:

- Отрицательное число, если в соответствии с вашим критерием сортировки *a* «меньше» *b* и должно присутствовать в отсортированном массиве перед *b*.
- Ноль, если *a* и *b* для целей сортировки эквивалентны.
- Положительное число, если для целей сортировки *a* «больше» *b*.

Заметьте, что неопределенные элементы при сортировке всегда оказываются в конце массива. Это происходит, даже если указана специальная функция сортировки: неопределенные значения никогда не передаются в заданную вами функцию *orderfunc*.

Пример

Следующий код показывает, как написать функцию сравнения, сортирующую массив чисел в числовом, а не в алфавитном порядке:

```
// Функция для сортировки чисел в порядке возрастания
function numberorder(a, b) { return a - b; }

a = new Array(33, 4, 1111, 222);
a.sort();             // Алфавитная сортировка: 1111, 222, 33, 4
a.sort(numberorder);  // Числовая сортировка: 4, 33, 222, 1111
```

Array.splice()

JavaScript 1.2; JScript 5.5; ECMAScript v3

вставляет, удаляет или заменяет элементы массива

Синтаксис

массив.splice(*начало*, *удаляемое_количество*, *значение*, ...)

Аргументы

начало

Элемент массива, с которого следует начать вставку или удаление.

удаляемое_количество

Количество элементов массива, начиная с элемента, заданного аргументом *начало*, и включая его, которые должны быть удалены из *массива*. Этот аргумент не является обязательным. Если он не указан, splice() удаляет все элементы от позиции, заданной аргументом *начало*, до конца массива.

значение, ...

Ноль или более значений, которые должны быть вставлены в *массив*, начиная с индекса, указанного в аргументе *начало*.

Возвращаемое значение

Массив, содержащий удаленные из *массива* элементы, если они есть. Из-за ошибки возвращаемое значение не всегда является массивом в реализации Netscape JavaScript 1.2.

Описание

splice() удаляет указанное количество элементов массива, начиная с элемента, позиция которого определяется аргументом *начало*, и включая его, и заменяет значениями, указанными в списке аргументов. Элементы массива, расположенные после вставляемых или удаляемых элементов, сдвигаются и образуют непрерывную последовательность с остальной частью массива. Заметьте, однако, что, в отличие от метода с похожим именем, slice(), splice() непосредственно изменяет *массив*.

Пример

Работу splice() проще всего понять на примере:

```
var a = [1,2,3,4,5,6,7,8]
a.splice(4);        // Возвращает [5,6,7,8]; a равно [1,2,3,4]
a.splice(1,2);      // Возвращает [2,3]; a равно [1,4]
a.splice(1,1);      // Netscape JavaScript 1.2 возвращает 4, а не [4]
a.splice(1,0,2,3);  // Netscape JavaScript 1.2 возвращает undefined, а не []
```

Ошибки

Предполагается, что `splice()` во всех случаях возвращает массив удаленных элементов. Однако в интерпретаторе Netscape JavaScript 1.2, когда удален один элемент, метод возвращает этот элемент, а не массив, содержащий этот элемент. Кроме того, если ни один элемент не удален, он не возвращает ничего, а не пустой массив. Реализации JavaScript от Netscape эмулируют это ошибочное поведение, когда явно указана версия языка 1.2.

Смотри также

Array.slice()

Array.toLocaleString()

JavaScript 1.5; JScript 5.5; ECMAScript v1

преобразует массив в локализованную строку **перекрывает Object.toLocaleString()**

Синтаксис

массив.`toLocaleString()`

Возвращаемое значение

Локализованное строковое представление *массива*.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод `toLocaleString()` массива возвращает локализованное строковое представление массива. Это делается путем вызова метода `toLocaleString()` для всех элементов массива и последующей конкатенации полученных строк с помощью символа-разделителя, определяемого региональными настройками.

Смотри также

Array.toString(), Object.toLocaleString()

Array.toString()

JavaScript 1.1; JScript 2.0; ECMAScript v1

преобразует массив в строку **перекрывает Object.toString()**

Синтаксис

массив.`toString()`

Возвращаемое значение

Строковое представление *массива*.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося массивом.

Описание

Метод `toString()` массива преобразует массив в строку и возвращает строку. Когда массив используется в строковом контексте, JavaScript автоматически преобразует

его в строку путем вызова этого метода. Однако в некоторых случаях может потребоваться явный вызов `toString()`.

`toString()` сначала преобразует в строку каждый элемент (вызывая их методы `toString()`). После преобразования все элементы выводятся в виде списка строк, разделенных запятыми. Это значение совпадает со значением, возвращаемым методом `join()` без аргументов.

Ошибки

В реализациях от Netscape, когда явно указана версия 1.2 языка, `toString()` возвращает заключенный в квадратные скобки список элементов массива, разделенных запятыми и пробелами в соответствии с нотацией литерала массива. Это, например, происходит, когда для атрибута `language` тега `<script>` явно задано значение «JavaScript1.2».

Смотри также

Array.toLocaleString(), Object.toString()

Array.unshift()

JavaScript 1.2; JScript 5.5; ECMAScript v3

вставляет элементы в начало массива

Синтаксис

массив.unshift(*значение*, ...)

Аргументы

значение, ...

Одно и более значений, которые должны быть вставлены в начало *массива*.

Возвращаемое значение

Новая длина массива.

Описание

`unshift()` вставляет свои аргументы в начало *массива*, сдвигая существующие элементы к верхним индексам для создания места. Первый аргумент `shift()` становится новым нулевым элементом массива, второй аргумент – новым первым элементом, и т. д. Обратите внимание, что `unshift()` не создает новый массив, а непосредственно изменяет *массив*.

Пример

`unshift()` часто применяется совместно с `shift()`. Например:

```
var a = [];            // a:[]
a.unshift(1);          // a:[1]          Возвращает: 1
a.unshift(22);         // a:[22,1]       Возвращает: 2
a.shift();             // a:[1]          Возвращает: 22
a.unshift(33,[4,5]);   // a:[33,[4,5],1] Возвращает: 3
```

Смотри также

Array.shift()

Boolean

JavaScript 1.1; JScript 2.0; ECMAScript v1

поддержка логических значений **наследуется от Object**

Конструктор

```
new Boolean(значение)  // функция-конструктор
Boolean(значение)      // функция преобразования
```

Аргументы

значение

Значение, которое должно быть сохранено в объекте Boolean или которое должно быть преобразовано в логическое значение.

Возвращаемое значение

При вызове в качестве конструктора (с оператором `new`) `Boolean()` преобразует аргумент в логическое значение и возвращает объект Boolean, содержащий это значение. При вызове в качестве функции (без оператора `new`) `Boolean()` просто преобразует свой аргумент в элементарное логическое значение и возвращает это значение.

Значения 0, NaN, `null`, литерал `false`, пустая строка `""` и `undefined` преобразуются в `false`. Все остальные элементарные значения (включая строку «false»), а также все объекты и массивы преобразуются в `true`.

Методы

`toString()`

Возвращает `true` или `false`, в зависимости от логического значения, представляемого объектом Boolean.

`valueOf()`

Возвращает элементарное логическое значение, содержащееся в объекте Boolean.

Описание

Логические значения – это базовый тип данных JavaScript. Объект Boolean представляет собой «обертку» вокруг логического значения. Объектный тип Boolean в основном существует для предоставления метода `toString()`, который преобразует логические значения в строки. Когда метод `toString()` вызывается для преобразования логического значения в строку (а часто он вызывается JavaScript неявно), JavaScript преобразует логическое значение во временный объект Boolean, для которого может быть вызван метод `toString()`.

Смотри также

Object

Boolean.toString()

JavaScript 1.1; JScript 2.0; ECMAScript v1

преобразует логическое значение в строку **перекрывает Object.toString()**

Синтаксис

b.`toString()`

Возвращаемое значение

Строка «true» или «false», в зависимости от элементарного логического значения или значения объекта Boolean *b*.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося Boolean.

Boolean.valueOf()

JavaScript 1.1; JScript 2.0; ECMAScript v1

логическое значение объекта Boolean **перекрывает Object.valueOf()**

Синтаксис

b`.valueOf()`

Возвращаемое значение

Элементарное логическое значение, содержащееся в объекте Boolean *b*.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося Boolean.

Date

JavaScript 1.0; JScript 1.0; ECMAScript v1

работа с датами и временем **наследуется от Object**

Конструктор

`new Date()`

`new Date(`*миллисекунды*`)`

`new Date(`*строка_даты*`)`

`new Date(`*год*`, `*месяц*`, `*день*`, `*часы*`, `*минуты*`, `*секунды*`, `*мс*`)`

Без указания аргументов конструктор `Date()` создает объект Date со значением, равным текущей дате и времени. Если конструктору передается единственный числовой аргумент, он используется как внутреннее числовое представление даты в миллисекундах, аналогичное возвращаемому методом `getTime()`. Когда передается один строковый аргумент, он рассматривается как строковое представление даты в формате, принимаемом методом `Date.parse()`. Кроме того, конструктору можно передать от двух до семи числовых аргументов, задающих индивидуальные поля даты и времени. Все аргументы, кроме первых двух – полей года и месяца, – могут отсутствовать. Обратите внимание, что эти поля даты и времени задаются на основе локального времени, а не времени UTC (аналогичного GMT). См. альтернативный статический метод `Date.UTC()`.

`Date()` может также вызываться как функция, без оператора `new`. При таком вызове `Date()` игнорирует любые переданные аргументы и возвращает текущую дату и время.

Аргументы

миллисекунды

Количество миллисекунд между нужной датой и полуночью 1 января 1970 года (UTC). Например, передав в качестве аргумента число 5000, мы создадим дату, обозначающую пять секунд после полуночи 1 января 1970 года.

строка_даты

Единственный аргумент, задающий дату и (необязательно) время в виде строки. Строка должна иметь формат, понятный для `Date.parse()`.

год

Год в виде четырех цифр. Например, `2001` для 2001 года. Для совместимости с более ранними реализациями JavaScript, если значение этого аргумента находится между 0 и 99, к нему добавляется 1900.

месяц

Месяц, заданный в виде целого от 0 (Январь) до 11 (Декабрь).

день

День месяца, заданный в виде целого от 1 до 31. Обратите внимание, что наименьшее из значений этого аргумента равно 1, а остальных аргументов – 0. Может отсутствовать.

часы

Часы, заданные в виде целого от 0 (полночь) до 23 (11 часов вечера). Необязательный аргумент.

минуты

Минуты в часах, указанные в виде целого от 0 до 59. Необязательный аргумент.

секунды

Секунды в минутах, указанные в виде целого от 0 до 59. Необязательный аргумент.

мс

Миллисекунды в секунде, указанные в виде целого от 0 до 999. Необязательный аргумент.

Методы

У объекта Date нет доступных для записи или чтения свойств; вместо этого доступ к значениям даты и времени выполняется через методы. Большинство методов объекта Date имеют две формы: одна для работы с локальным временем, а другая с универсальным временем (UTC или GMT). Если в имени метода присутствует строка «UTC», он работает с универсальным временем. Эти пары методов указываются в приведенном ниже списке вместе. Например, обозначение `get[UTC]Day()` относится к двум методам: `getDay()` и `getUTCDay()`.

Методы объекта Date могут вызываться только для объектов с типом Date и генерируют исключение TypeError, если вызывать их для объектов другого типа.

`get[UTC]Date()`

Возвращает день месяца из объекта Date в соответствии с локальным или универсальным временем.

`get[UTC]Day()`

Возвращает день недели из объекта Date в соответствии с локальным или универсальным временем.

`get[UTC]FullYear()`

Возвращает год даты в полном четырехзначном формате, в локальном или универсальном времени.

`get[UTC]Hours()`

Возвращает поле часов в объекте Date в локальном или универсальном времени.

`get[UTC]Milliseconds()`

Возвращает поле миллисекунд в объекте Date в локальном или универсальном времени.

`get[UTC]Minutes()`

Возвращает поле минут в объекте Date в локальном или универсальном времени.

`get[UTC]Month()`

Возвращает поле месяца в объекте Date в локальном или универсальном времени.

`get[UTC]Seconds()`

Возвращает поле секунд в объекте Date в локальном или универсальном времени.

`getTime()`

Возвращает внутреннее представление (миллисекунды) объекта Date. Обратите внимание, что это значение не зависит от часового пояса, и, следовательно, отсутствует отдельный метод `getUTCTime()`.

`getTimezoneOffset()`

Возвращает разницу (в минутах) между локальным и универсальным представлениями этой даты. Обратите внимание, что возвращаемое значение зависит от того, действует ли летнее время для указанной даты.

`getYear()`

Возвращает поле года в объекте Date. Признан устаревшим, рекомендуется вместо него применять метод `getFullYear()`.

`set[UTC]Date()`

Устанавливает день месяца в Date в соответствии с локальным или универсальным временем.

`set[UTC]FullYear()`

Устанавливает год (и, возможно, месяц и день) в Date в соответствии с локальным или универсальным временем.

`set[UTC]Hours()`

Устанавливает час (и, возможно, поля минут, секунд и миллисекунд) в Date в соответствии с локальным или универсальным временем.

`set[UTC]Milliseconds()`

Устанавливает поле миллисекунд в Date в соответствии с локальным или универсальным временем.

`set[UTC]Minutes()`

Устанавливает поле минут (и, возможно, поля секунд и миллисекунд) в Date в соответствии с локальным или универсальным временем.

`set[UTC]Month()`

Устанавливает поле месяца (и, возможно, дня месяца) в Date в соответствии с локальным или универсальным временем.

`set[UTC]Seconds()`

Устанавливает поле секунд (и, возможно, поле миллисекунд) в Date в соответствии с локальным или универсальным временем.

`setTime()`

Устанавливает поля объекта Date в соответствии с миллисекундным форматом.

`setYear()`

Устанавливает поле года объекта Date. Признан устаревшим, вместо него рекомендуется использовать `setFullYear()`.

`toDateString()`

Возвращает строку, представляющую дату из Date для локального часового пояса.

`toGMTString()`

Преобразует Date в строку, беря за основу часовой пояс GMT. Признан устаревшим, вместо него рекомендован `toUTCString()`.

`toLocaleDateString()`

Возвращает строку, представляющую дату из Date в локальном часовом поясе, в соответствии с локальными соглашениями по форматированию дат.

`toLocaleString()`

Преобразует Date в строку в соответствии с локальным часовым поясом и локальными соглашениями о форматировании дат.

`toLocaleTimeString()`

Возвращает строку, представляющую время из Date в локальном часовом поясе на основе локальных соглашений о форматировании времени.

`toString()`

Преобразует Date в строку в соответствии с локальным часовым поясом.

`toTimeString()`

Возвращает строку, представляющую время из Date в локальном часовом поясе.

`toUTCString()`

Преобразует Date в строку, используя универсальное время.

`valueOf()`

Преобразует Date в его внутренний миллисекундный формат.

Статические методы

В дополнение ко многим методам экземпляра, перечисленным выше, в объекте Date также определены два статических метода. Эти методы вызываются через сам конструктор `Date()`, а не через индивидуальные объекты Date:

`Date.parse()`

Анализирует строковое представление даты и времени и возвращает внутреннее представление этой даты в миллисекундах.

`Date.UTC()`

Возвращает миллисекундное представление указанной даты и времени UTC.

Описание

Объект Date – это тип данных, встроенный в язык JavaScript. Объекты Date создаются с помощью синтаксиса `new Date()`, показанного в разделе «Конструктор».

После создания объекта Date можно воспользоваться многочисленными методами для работы с ним. Многие из методов позволяют получить и установить поля года, месяца, дня, часа, минуты, секунды и миллисекунды в соответствии либо с локальным временем, либо с временем UTC (универсальным, или GMT). Метод `toString()` и его варианты преобразуют даты в читаемые строки. `getTime()` и `setTime()` преобразуют количество миллисекунд, прошедших с полуночи (GMT) 1 января 1970 года, во внутреннее представление объекта Date и из него. В этом стандартном миллисекундном формате дата и время представляются одним целым, что делает дату очень простой арифметически. Стандарт ECMAScript требует, чтобы объект Date мог представлять любую дату и время, с миллисекундной точностью в пределах 100 миллионов дней до и после 01.01.1970. Этот диапазон равен ±273 785 лет, поэтому часы JavaScript будут правильно работать до 275 755 года.

Пример

Известно множество методов, позволяющих работать с созданным объектом Date:

```
d = new Date();  // Получает текущую дату и время
document.write('Сегодня: " + d.toLocaleDateString() + '. ');  // Показывает дату
document.write('Время: ' + d.toLocaleTimeString());           // Показывает время
var dayOfWeek = d.getDay();                              // Какой сегодня день недели?
var weekend = (dayOfWeek == 0) || (dayOfWeek == 6);  // Сегодня выходной?
```

Еще одно обычное применение объекта Date – это вычитание миллисекундного представления текущего времени из другого времени для определения относительного местоположения двух временных меток. Следующий клиентский пример показывает два таких применения:

```
<script language="JavaScript">
today = new Date();      // Запоминаем сегодняшнюю дату
christmas = new Date();  // Получаем дату из текущего года
christmas.setMonth(11);  // Устанавливаем месяц декабрь...
christmas.setDate(25);   // и 25-е число

// Если Рождество еще не прошло, вычисляем количество миллисекунд между
// текущим моментом и Рождеством, преобразуем его в количество дней
// и печатаем сообщение
if (today.getTime() < christmas.getTime()) {
    difference = christmas.getTime() - today.getTime();
    difference = Math.floor(difference / (1000 * 60 * 60 * 24));
    document.write('Всего ' + difference + ' дней до Рождества!<p>');
}
</script>

// ... остальная часть HTML-документа ...

<script language="JavaScript">
// Здесь мы используем объекты Date для измерения времени
// Делим на 1000 для преобразования миллисекунд в секунды
now = new Date();
document.write('<p>Страница загружалась' +
    (now.getTime()-today.getTime())/1000 +
    'секунд.');
</script>
```

Смотри также

Date.parse(), Date.UTC()

Date.getDate()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает день месяца

Синтаксис

дата.getDate()

Возвращаемое значение

День месяца указанного объекта Date *дата* в соответствии с локальным временем. Возвращаемые значения могут находиться в интервале между 1 и 31.

Date.getDay()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает день недели

Синтаксис

дата.getDay()

Возвращаемое значение

День недели в указанном объекте Date *дата* в соответствии с локальным временем. Возвращает числа от 0 (воскресенье) до 6 (суббота).

Date.getFullYear()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает год

Синтаксис

дата.getFullYear()

Возвращаемое значение

Год, получаемый, когда *дата* выражена в локальном времени. Возвращает четыре цифры, а не сокращение из двух цифр.

Date.getHours()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает поле часов объекта Date

Синтаксис

дата.getHours()

Возвращаемое значение

Значение поля часов в локальном времени в указанном объекте Date *дата*. Возвращаемое значение находится в диапазоне между 0 (полночь) и 23 (11 часов вечера).

Date.getMilliseconds()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает поле миллисекунд в Date

Синтаксис

дата.getMilliseconds()

Возвращаемое значение

Поле миллисекунд в объекте *дата*, вычисленное в локальном времени.

Date.getMinutes()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает поле минут в Date

Синтаксис

дата.getMinutes()

Возвращаемое значение

Поле минут, вычисленное в локальном времени, в указанном объекте Date *дата*. Возвращаемое значение может принимать значения от 0 до 59.

Date.getMonth()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает месяц для Date

Синтаксис

дата.getMonth()

Возвращаемое значение

Поле месяца, вычисленное в локальном времени, в указанном объекте Date *дата*. Возвращаемое значение может принимать значения от 0 (январь) до 11 (декабрь).

Date.getSeconds()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает поле секунд для Date

Синтаксис

дата.getSeconds()

Возвращаемое значение

Поле секунд, вычисленное в локальном времени, в указанном объекте Date *дата*. Возвращаемое значение может принимать значения от 0 до 59.

Date.getTime()

JavaScript 1.0; JScript 1.0; ECMAScript v1

возвращает Date в миллисекундах

Синтаксис

дата.getTime()

Возвращаемое значение

Миллисекундное представление указанного объекта Date *дата*; то есть число миллисекунд между полуночью 01.01.1970 и датой/временем, определяемыми *дата*.

Описание

getTime() преобразует дату и время в одно целое значение. Это полезно, когда требуется сравнить два объекта Date или определить время, прошедшее между двумя датами. Обратите внимание, что миллисекундное представление даты не зависит от часового пояса, поэтому отсутствует метод getUTCTime(), дополняющий данный. Не путайте метод getTime() с методами getDay() и getDate(), возвращающими соответственно день недели и день месяца.

Метод Date.parse() и Date.UTC() позволяют преобразовать спецификацию даты и времени в миллисекундное представление, обходя избыточное создание объекта Date.

Смотри также

Date, Date.parse(), Date.setTime(), Date.UTC()

Date.getTimezoneOffset() JavaScript 1.0; JScript 1.0; ECMAScript v1

определяет смещение от GMT

Синтаксис

дата.getTimezoneOffset()

Возвращаемое значение

Разница в минутах между временем по Гринвичу (GMT) и локальным временем.

Описание

Функция getTimezoneOffset() возвращает разницу в минутах между универсальным и локальным временем. Сообщает, в каком часовом поясе исполняется код JavaScript и действует ли (или будет ли действовать) летнее время для указанной *даты*.

Возвращаемое значение измеряется в минутах, а не в часах, поскольку в некоторых странах имеются часовые пояса, не занимающие целого часового интервала.

Date.getUTCDate() JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает день месяца (универсальное время)

Синтаксис

дата.getUTCDate()

Возвращаемое значение

День месяца (значение между 1 и 31), полученное при вычислении *даты* в универсальном времени.

Date.getUTCDay() JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает день месяца (универсальное время)

Синтаксис

дата.getUTCDay()

Возвращаемое значение

День недели, получаемый, когда *дата* выражена в универсальном времени. Возвращаемые значения могут находиться в интервале между 0 (воскресенье) и 6 (суббота).

Date.getUTCFullYear()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает год (универсальное время)

Синтаксис

дата.getUTCFullYear()

Возвращаемое значение

Год, получаемый, когда *дата* вычисляется в универсальном времени. Возвращаемое значение – четырехзначный номер года, а не сокращение из двух цифр.

Date.getUTCHours()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает поле часов для Date (универсальное время)

Синтаксис

дата.getUTCHours()

Возвращаемое значение

Поле часов для *даты*, вычисленное в универсальном времени. Возвращаемое значение – целое между 0 (полночь) и 23 (11 часов вечера).

Date.getUTCMilliseconds()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает поле миллисекунд для Date (универсальное время)

Синтаксис

дата.getUTCMilliseconds()

Возвращаемое значение

Поле миллисекунд для *даты*, выраженное в универсальном времени.

Date.getUTCMinutes()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает поле минут для Date (универсальное время)

Синтаксис

дата.getUTCMinutes()

Возвращаемое значение

Поле минут для *даты*, в универсальном времени. Возвращает целое между 0 и 59.

Date.getUTCMonth()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает месяц года (универсальное время)

Синтаксис

дата.getUTCMonth()

Возвращаемое значение

Месяц года, получающийся, когда *дата* вычислена в универсальном времени. Возвращает целое между 0 (январь) и 11 (декабрь). Обратите внимание, что объект Date обозначает первый день месяца цифрой 1, но первому месяцу года соответствует цифра 0.

Date.getUTCSeconds()

JavaScript 1.2; JScript 3.0; ECMAScript v1

возвращает поле секунд для Date (универсальное время)

Синтаксис

дата.getUTCSeconds()

Возвращаемое значение

Поле секунд *даты*, в универсальном времени. Возвращает целое между 0 и 59.

Date.getYear()

JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

возвращает поле года для Date

Синтаксис

дата.getYear()

Возвращаемое значение

Поле года для указанного объекта Date *дата* минус 1900.

Описание

getYear() возвращает поле года для указанного объекта Date минус 1900. По спецификации ECMAScript v3 этот метод не является обязательным в совместимых реализациях JavaScript; используйте вместо него getFullYear().

Ошибки

Реализации Netscape JavaScript от 1.0 до 1.2 вычитают 1900 только для годов между 1900 и 1999.

Date.parse()

JavaScript 1.0; JScript 1.0; ECMAScript v1

синтаксический анализ строки даты/времени

Синтаксис

Date.parse(*дата*)

Аргументы

дата

Строка для разбора, содержащая дату и время.

Возвращаемое значение

Количество миллисекунд между указанной датой/временем и полуночью 1 января 1970 года по Гринвичу.

Описание

Date.parse() — это статический метод объекта Date. Он всегда вызывается через конструктор Date как Date.parse(), а не через объект Date как *дата*.parse(). Метод Date.parse() принимает один строковый аргумент. Он анализирует дату, содержащуюся в строке, и возвращает ее в миллисекундном формате, который может использоваться непосредственно, для создания нового объекта Date или для установки даты в существующем объекте Date с помощью Date.setTime().

Стандарт ECMAScript не определяет формат строк, которые могут быть разобраны `Date.parse()`, за исключением того, что метод может разбирать строки, возвращаемые методами `Date.toString()` и `Date.toUTCString()`. К сожалению, эти функции форматируют даты зависимым от реализации образом, поэтому нет универсального способа написания дат, гарантированно понятного любым реализациям JavaScript.

Смотри также

Date, Date.setTime(), Date.toGMTString(), Date.UTC()

Date.setDate()

JavaScript 1.0; JScript 1.0; ECMAScript v1

устанавливает день месяца

Синтаксис

дата.setDate(*день_месяца*)

Аргументы

день_месяца

Целое между 1 и 31, выступающее в качестве нового значения (в локальном времени) поля «день месяца» объекта *дата*.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод ничего не возвращал.

Date.setFullYear()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает год и, возможно, месяц и день

Синтаксис

дата.setFullYear(*год*)

дата.setFullYear(*год*, *месяц*)

дата.setFullYear(*год*, *месяц*, *день*)

Аргументы

год

Год, выраженный в локальном времени, который должен быть установлен в *дате*. Этот аргумент должен быть целым, включающим век, например 1999; не может быть сокращением, таким как 99.

месяц

Необязательное целое между 0 и 11, используемое для установки нового значения поля месяца (в локальном времени) для *даты*.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля «день месяца» для *даты* (в локальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setHours()

JavaScript 1.0; JScript 1.0; ECMAScript v1

устанавливает поля часов, минут, секунд и миллисекунд в Date

Синтаксис

дата.setHours(*часы*)

дата.setHours(*часы*, *минуты*)

дата.setHours(*часы*, *минуты*, *секунды*)

дата.setHours(*часы*, *минуты*, *секунды*, *миллисекунды*)

Аргументы

часы

Целое между 0 (полночь) и 23 (11 часов вечера) локального времени, устанавливаемое в качестве нового значения часов в *дате*.

минуты

Необязательное целое, между 0 и 59, используемое в качестве нового значения поля минут в *дате* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

секунды

Необязательное целое, между 0 и 59. Представляет собой новое значение поля секунд в *дате* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

миллисекунды

Необязательное целое, между 0 и 999, выступающее в качестве нового значения поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод не возвращал ничего.

Date.setMilliseconds()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поле миллисекунд в Date

Синтаксис

дата.setMilliseconds(*миллисекунды*)

Аргументы

миллисекунды

Поле миллисекунд, выраженное в локальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Миллисекундное представление измененной даты.

Date.setMinutes()

JavaScript 1.0; JScript 1.0; ECMAScript v1

Устанавливает поля минут и секунд в Date

Синтаксис

дата.setMinutes(*минуты*)

дата.setMinutes(*минуты*, *секунды*)

дата.setMinutes(*минуты*, *секунды*, *миллисекунды*)

Аргументы

минуты

Целое между 0 и 59, устанавливаемое в качестве значения минут (в локальном времени) в объекте Date *дата*.

секунды

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля секунд *даты* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

миллисекунды

Необязательное целое между 0 и 999, представляет собой новое значение (в локальном времени) поля миллисекунд *даты*. Этот аргумент не поддерживался до стандартизации ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод не возвращал ничего.

Date.setMonth()

JavaScript 1.0; JScript 1.0; ECMAScript v1

устанавливает поля месяца и дня в Date

Синтаксис

дата.setMonth(*месяц*)

дата.setMonth(*месяц*, *день*)

Аргументы

месяц

Целое между 0 (январь) и 11 (декабрь), устанавливаемое в качестве значения месяца для объекта Date *дата* (в локальном времени). Обратите внимание, что месяцы нумеруются начиная с 0, а дни в месяце – с 1.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля «день месяца» в *дате* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод не возвращал ничего.

Date.setSeconds()

JavaScript 1.0; JScript 1.0; ECMAScript v1

устанавливает поля секунд и миллисекунд в Date

Синтаксис

дата.setSeconds(*секунды*)

дата.setSeconds(*секунды*, *миллисекунды*)

Аргументы

секунды

Целое между 0 и 59, устанавливаемое как значение секунд в объекте Date *дата*.

миллисекунды

Необязательное целое между 0 и 999, выступающее в качестве нового значения поля миллисекунд в *дате* (в локальном времени). Этот аргумент не поддерживался до стандартизации ECMAScript.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод не возвращал ничего.

Date.setTime()

JavaScript 1.0; JScript 1.0; ECMAScript v1

устанавливает Date в миллисекундах

Синтаксис

дата.setTime(*миллисекунды*)

Аргументы

миллисекунды

Количество миллисекунд между требуемой датой/временем и полуночью по Гринвичу 1 января 1970 года. Подобное миллисекундное значение может быть также передано конструктору Date() и может быть получено при вызове методов Date.UTC() и Date.parse(). Представление даты в миллисекундном формате делает ее независимой от часового пояса.

Возвращаемое значение

Аргумент *миллисекунды*. До стандартизации ECMAScript метод не возвращал ничего.

Date.setUTCDate()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает день месяца (универсальное время)

Синтаксис

дата.setUTCDate(*день_месяца*)

Аргументы

день_месяца

День месяца, выраженный в универсальном времени, устанавливаемый в *дате*. Этот аргумент должен быть целым между 1 и 31.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCFullYear() JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает год, месяц и день (универсальное время)

Синтаксис

дата.setUTCFullYear(*год*)

дата.setUTCFullYear(*год*, *месяц*)

дата.setUTCFullYear(*год*, *месяц*, *день*)

Аргументы

год

Год, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым, включающим век, например 1999, а не сокращением, как 99.

месяц

Необязательное целое между 0 и 11, выступающее в качестве нового значения в поле месяца *даты* (в универсальном времени).

день

Необязательное целое между 1 и 31, представляет собой новое значение (в универсальном времени) поля «день месяца» в *дате*.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCHours() JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поля часов, минут, секунд и миллисекунд в Date (универсальное время)

Синтаксис

дата.setUTCHours(*часы*)

дата.setUTCHours(*часы*, *минуты*)

дата.setUTCHours(*часы*, *минуты*, *секунды*)

дата.setUTCHours(*часы*, *минуты*, *секунды*, *миллисекунды*)

Аргументы

часы

Поле «часы», выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (полночь) и 23 (11 часов вечера).

минуты

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля минут в *дате* (в универсальном времени).

секунды

Необязательное целое между 0 и 59, представляет собой новое значение (в универсальном времени) поля секунд в *дате*.

миллисекунды

Необязательное целое между 0 и 999, используемое в качестве нового значения для поля миллисекунд в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMilliseconds()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поле миллисекунд в Date (универсальное время)

Синтаксис

дата.setUTCMilliseconds(*миллисекунды*)

Аргументы

миллисекунды

Поле миллисекунд, выраженное в универсальном времени, которое должно быть установлено в *дате*. Этот аргумент должен быть целым между 0 и 999.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMinutes()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поля минут и секунд в Date (универсальное время)

Синтаксис

дата.setUTCMinutes(*минуты*)

дата.setUTCMinutes(*минуты*, *секунды*)

дата.setUTCMinutes(*минуты*, *секунды*, *миллисекунды*)

Аргументы

минуты

Поле минут, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен принимать значение между 0 и 59.

секунды

Необязательное целое между 0 и 59, выступающее в качестве нового значения поля секунд в *дате* (в универсальном времени).

миллисекунды

Необязательное целое между 0 и 999, представляет собой новое значение (в универсальном времени) поля миллисекунд в *дате*.

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCMonth()

JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поля месяца и дня в Date (универсальное время)

Синтаксис

дата.setUTCMonth(*месяц*)

дата.setUTCMonth(*месяц*, *день*)

Аргументы

месяц

Месяц, выраженный в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 (январь) и 11 (декабрь). Обратите внимание, что месяцы нумеруются начиная с 0, а дни в месяце – с 1.

день

Необязательное целое между 1 и 31, выступающее в качестве нового значения поля «день месяца» в *дате* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setUTCSeconds() JavaScript 1.2; JScript 3.0; ECMAScript v1

устанавливает поля секунд и миллисекунд в Date (универсальное время)

Синтаксис

дата.setUTCSeconds(*секунды*)

дата.setUTCSeconds(*секунды*, *миллисекунды*)

Аргументы

секунды

Поле секунд, выраженное в универсальном времени, для установки в *дате*. Этот аргумент должен быть целым между 0 и 59.

миллисекунды

Необязательное целое между 0 и 999, используемое в качестве нового значения поля миллисекунд *даты* (в универсальном времени).

Возвращаемое значение

Внутреннее миллисекундное представление измененной даты.

Date.setYear() JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

устанавливает поле года в Date

Синтаксис

дата.setYear(*год*)

Аргументы

год

Целое, устанавливаемое в качестве значения года (в локальном времени) для объекта Date *дата*. Если это значение находится между 0 и 99, к нему добавляется 1900, и оно рассматривается как год между 1900 и 1999.

Возвращаемое значение

Миллисекундное представление измененной даты. До стандартизации ECMAScript этот метод не возвращал ничего.

Описание

`setYear()` устанавливает поле года в указанном объекте Date со специальным поведением для годов между 1900 и 1999.

По спецификации ECMAScript v3 этот метод не является обязательным в совместимых реализациях JavaScript; вместо него рекомендован `setFullYear()`.

Date.toDateString() JavaScript 1.5; JScript 5.5; ECMAScript v3

возвращает дату из Date в виде строки

Синтаксис

дата.`toDateString()`

Возвращаемое значение

Зависящее от реализации понятное человеку строковое представление даты (без времени) из объекта *дата*, выраженное в локальном времени.

Смотри также

Date.toLocaleDateString(), Date.toLocaleString(), Date.toLocaleTimeString(), Date. toString(), Date.toTimeString()

Date.toGMTString() JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

преобразует Date в строку универсального времени

Синтаксис

дата.`toGMTString()`

Возвращаемое значение

Строковое представление даты и времени, заданной объектом Date *дата*. Перед преобразованием в строку дата переводится из локального часового пояса в гринвичский.

Описание

`toGMTString()` признан устаревшим, рекомендуется применять аналогичный метод `Date.toUTCString()`.

По спецификации ECMAScript v3 совместимые реализации JavaScript больше не обязаны предоставлять этот метод; используйте вместо него `toUTCString()`.

Смотри также

Date.toUTCString()

Date.toLocaleDateString() JavaScript 1.5; JScript 5.5; ECMAScript v3

возвращает дату из Date в виде строки в локальном формате

Синтаксис

дата.`toLocaleDateString()`

Возвращаемое значение

Зависящее от реализации понятное человеку строковое представление даты (без времени) из объекта *дата*, выраженное в локальном часовом поясе и отформатированное в соответствии с локальными соглашениями.

Смотри также

Date.toDateString(), Date.toLocaleString(), Date.toLocaleTimeString(), Date.toString(), Date.toTimeString()

Date.toLocaleString() JavaScript 1.0; JScript 1.0; ECMAScript v1

преобразует Date в строку в локальном формате

Синтаксис

дата.toLocaleString()

Возвращаемое значение

Строковое представление даты и времени, заданных *датой*. Дата и время представлены в локальном часовом поясе и отформатированы в соответствии с региональными соглашениями.

Применение

toLocaleString() преобразует дату в строку в соответствии с локальным часовым поясом. Этот метод также руководствуется локальными соглашениями для форматирования даты и времени, поэтому формат может отличаться на разных платформах и в разных странах. toLocaleString() возвращает строку, отформатированную в соответствии с возможно предпочтительным для пользователя форматом даты и времени.

Смотри также

Date.toLocaleDateString(), Date.toLocaleTimeString(), Date.toString(), Date.toUTCString()

Date.toLocaleTimeString() JavaScript 1.5; JScript 5.5; ECMAScript v3

возвращает данные о времени из Date в виде строки в локальном формате

Синтаксис

дата.toLocaleTimeString()

Возвращаемое значение

Зависящее от реализации понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе и отформатированное в соответствии с локальными соглашениями.

Смотри также

Date.toDateString(), Date.toLocaleDateString(), Date.toLocaleString(), Date.toString(), Date.toTimeString()

Date.toString()

JavaScript 1.0; JScript 1.0; ECMAScript v1

преобразует дату в строку **перекрывает Object.toString()**

Синтаксис

дата.toString()

Возвращаемое значение

Понятное человеку строковое представление *даты* в локальном часовом поясе.

Описание

toString() возвращает понятное человеку зависящее от реализации строковое представление *даты*. В отличие от toUTCString(), toString() вычисляет дату в локальном часовом поясе. В отличие от toLocaleString(), toString() может представлять дату и время, не используя форматирование, специфическое для региона.

Смотри также

Date.parse(), Date.toDateString(), Date.toLocaleString(), Date.toTimeString(), Date. toUTCString()

Date.toTimeString()

JavaScript 1.5; JScript 5.5; ECMAScript v3

возвращает данные о времени из Date в виде строки

Синтаксис

дата.toTimeString()

Возвращаемое значение

Зависящее от реализации понятное человеку строковое представление данных о времени из объекта *дата*, выраженное в локальном часовом поясе.

Смотри также

Date.toString(), Date.toDateString(), Date.toLocaleDateString(), Date.toLocaleString(), Date.toLocaleTimeString()

Date.toUTCString()

JavaScript 1.2; JScript 3.0; ECMAScript v1

преобразование Date в строку (универсальное время)

Синтаксис

дата.toUTCString()

Возвращаемое значение

Понятное человеку строковое представление *даты*, выраженное в универсальном времени.

Описание

toUTCString() возвращает зависящую от реализации строку, представляющую *дату* в универсальном времени.

Смотри также

Date.toLocaleString(), Date.toString()

Date.UTC()

JavaScript 1.0; JScript 1.0; ECMAScript v1

преобразует спецификацию даты в миллисекунды

Синтаксис

`Date.UTC(`*год*`,` *месяц*`,` *день*`,` *часы*`,` *минуты*`,` *секунды*`,` *мс*`)`

Аргументы

год

Год в четырехзначном формате. Если аргумент находится между 0 и 99, к нему добавляется 1900, и он рассматривается как год между 1900 и 1999.

месяц

Месяц, заданный в виде целого от 0 (январь) до 11 (декабрь).

день

День месяца, заданный в виде целого от 1 до 31. Обратите внимание, что наименьшее значение этого аргумента равно 1, наименьшее значение других аргументов – 0. Аргумент может отсутствовать.

часы

Час, заданный в виде целого от 0 (полночь) до 23 (11 часов вечера). Этот аргумент может отсутствовать.

минуты

Минуты в часе, заданные в виде целого от 0 до 59. Этот аргумент не является обязательным.

секунды

Секунды в минутах, заданные в виде целого от 0 до 59. Может отсутствовать.

мс

Количество миллисекунд. Этот аргумент не является обязательным и игнорировался до стандартизации ECMAScript.

Возвращаемое значение

Миллисекундное представление указанного универсального времени. Этот метод возвращает количество миллисекунд между полуночью по Гринвичу 1 января 1970 года и указанным временем.

Описание

`Date.UTC()` – это статический метод; он вызывается через конструктор `Date()`, а не через индивидуальный объект Date.

Аргументы `Date.UTC()` задают дату и время и подразумевают время в UTC (Universal Coordinated Time) – универсальное. Указанное время UTC преобразуется в миллисекундный формат, который может использоваться методом-конструктором `Date()` и методом `Date.setTime()`.

Метод-конструктор `Date()` может принимать аргументы даты и времени, идентичные тем, что принимает метод `Date.UTC()`. Разница в том, что конструктор `Date()` подразумевает локальное время, а `Date.UTC()` – время по Гринвичу (GMT). Создать объект Date, используя спецификацию времени в UTC, можно посредством следующего кода:

```
d = new Date(Date.UTC(1996, 4, 8, 16, 30));
```

Смотри также

Date, Date.parse(), Date.setTime()

Date.valueOf()

JavaScript 1.1; ECMAScript v1

преобразует Date в миллисекундное представление **перекрывает Object.valueOf()**

Синтаксис

дата.valueOf()

Возвращаемое значение

Миллисекундное представление *даты*. Возвращаемое значение совпадает со значением, возвращаемым Date.getTime().

decodeURI()

JavaScript 1.5; JScript 5.5; ECMAScript v3

декодирует символы в URI

Синтаксис

decodeURI(*uri*)

Аргументы

uri

Строка закодированного URI или другого текста, подлежащего декодированию.

Возвращаемое значение

Копия *uri*, в которой все шестнадцатеричные управляющие последовательности заменены на символы, которые они представляют.

Исключения

URIError

Означает, что одна или несколько управляющих последовательностей в *uri* имеют неверный формат и не могут быть правильно декодированы.

Описание

decodeURI() — это глобальная функция, возвращающая декодированную копию аргумента *uri*. Она выполняет действие, обратное действию, выполняемому функцией encodeURI(); подробности см. в описании этой функции.

Смотри также

decodeURIComponent(), encodeURI(), encodeURIComponent(), escape(), unescape()

decodeURIComponent()

JavaScript 1.5; JScript 5.5; ECMAScript v3

расшифровка управляющих последовательностей в компоненте URI

Синтаксис

decodeURIComponent(*s*)

Аргументы

s Строка, содержащая закодированную компоненту URI или другой текст, который должен быть декодирован.

Возвращаемое значение

Копия аргумента *s*, в котором шестнадцатеричные управляющие последовательности заменены символами, которые они представляют.

Исключения

URIError

Обозначает, что одна или несколько управляющих последовательностей в *uri* имеют неверный формат и не могут быть правильно декодированы.

Описание

decodeURIComponent() — глобальная функция, возвращающая декодированную копию своего аргумента *s*. Ее действие обратно кодированию, выполняемому функцией encodeURIComponent(). Подробности см. в справочной статье по этой функции.

Смотри также

decodeURI(), encodeURI(), encodeURIComponent(), escape(), unescape()

encodeURI()

JavaScript 1.5; JScript 5.5; ECMAScript v3

выполняет кодирование URI с помощью управляющих последовательностей

Синтаксис

encodeURI(*uri*)

Аргументы

uri

Строка, содержащая URI или другой текст, который должен быть закодирован.

Возвращаемое значение

Копия *uri*, в которой некоторые символы заменены на шестнадцатеричные управляющие последовательности.

Исключения

URIError

Указывает, что *uri* содержит искусственные пары Unicode неверного формата и не может быть закодирован.

Описание

encodeURI() — это глобальная функция, возвращающая закодированную копию аргумента *uri*. Не кодируются символы, цифры и следующие знаки пунктуации из ASCII:

```
- _ . ! ~ * ' ( )
```

Функция encodeURI() кодирует URI целиком, поэтому следующие символы пунктуации, имеющие в URI специальное значение, также не кодируются:

```
; / ? : @ & = + $ , #
```

Любые другие символы в *uri* заменяются путем преобразования символа в его код UTF-8 и последующего кодирования каждого из полученных байтов шестнадцатеричной управляющей последовательностью в формате %xx. В этой схеме кодирования ASCII-символы заменяются одной последовательностью %xx, символы с кода \u0080 до \u07ff заменяются двумя управляющими последовательностями, а все остальные 16-битовые символы Unicode заменяются тремя управляющими последовательностями.

Применяя этот метод для кодирования URI, необходимо быть уверенным, что ни один из компонентов URI (например, строка запроса) не содержит символов-разделителей URI, таких как ? и #. Если компоненты могут содержать эти символы, необходимо кодировать каждую компоненту отдельно с помощью encodeURIComponent().

Метод decodeURI() предназначен для выполнения действия, обратного кодированию. До ECMAScript v3 с помощью методов escape() и unescape() (сейчас признанных устаревшими) можно выполнять сходное кодирование и декодирование.

Пример

```
// Возвращает http://www.isp.com/app.cgi?arg1=1&arg2=hello%20world
encodeURI("http://www.isp.com/app.cgi?arg1=1&arg2=hello world");
encodeURI("\u00a9");  // Символ копирайта кодируется в %C2%A9
```

Смотри также

decodeURI(), decodeURIComponent(), encodeURIComponent(), escape(), unescape()

encodeURIComponent() JavaScript 1.5; JScript 5.5; ECMAScript v3

кодирует символы в компоненте URI в управляющие последовательности

Синтаксис

encodeURIComponent(*s*)

Аргументы

s – строка, содержащая фрагмент URI или другой текст, подлежащий кодированию.

Возвращаемое значение

Копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Исключения

URIError

Указывает, что *s* содержит искусственные пары Unicode неверного формата и не может быть закодирован.

Описание

encodeURIComponent() – глобальная функция; возвращает закодированную копию своего аргумента *s*. Не кодируются буквы, цифры и следующие знаки пунктуации из ASCII:

```
- _ . ! ~ * ' ( )
```

Все остальные символы, в том числе такие символы пунктуации, как /, :, #, служащие для разделения различных компонентов URI, заменяются на одну или несколько шестнадцатеричных управляющих последовательностей. Описание используемой схемы кодирования см. в статье о функции encodeURI().

Обратите внимание на разницу между encodeURIComponent() и encodeURI(): encodeURIComponent() предполагает, что его аргументом является фрагмент URI (такой как протокол, имя хоста, путь или строка запроса). Поэтому она преобразует символы пунктуации, используемые для разделения фрагментов URI.

Пример

```
encodeURIComponent("hello world?");  // Возвращает hello%20world%3F
```

Смотри также

decodeURI(), decodeURIComponent(), encodeURI(), escape(), unescape()

Error

JavaScript 1.5; JScript 5.5; ECMAScript v3

обобщенное исключение **наследуется от Object**

Конструктор

`new Error()`

`new Error(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении.

Возвращаемое значение

Только что созданный объект Error. Если задан аргумент *сообщение*, объект Error будет использовать его в качестве значения своего свойства `message`; в противном случае он возьмет в качестве значения этого свойства строку по умолчанию, определенную реализацией. Когда конструктор `Error()` вызывается как функция, без оператора `new`, он ведет себя так же, как при вызове с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. В этом свойстве хранится строка, переданная конструктору, или строка по умолчанию, определенная реализацией.

`name`

Строка, задающая тип исключения. Для экземпляров класса Error и всех его подклассов это свойство задает имя конструктора, с помощью которого был создан экземпляр.

Методы

`toString()`

Возвращает строку, определенную в реализации, которая представляет этот объект Error.

Описание

Экземпляры класса Error представляют ошибки или исключения и обычно используются с инструкциями `throw` и `try/catch`. Свойство `name` задает тип исключения, а посредством свойства `message` можно создать сообщение пользователю с подробной информацией об исключении.

Интерпретатор JavaScript никогда непосредственно не создает объект Error. Вместо этого он создает экземпляры одного из подклассов Error, таких как SyntaxError или RangeError. В вашем собственном коде для предупреждения об исключении может оказаться удобнее создавать объекты Error или просто выдавать сообщение об ошибке или ее код в виде элементарного строкового или числового значения.

Заметьте, что спецификация ECMAScript определяет для класса Error метод toString() (он наследуется всеми подклассами Error), но не требует, чтобы этот метод возвращал строку, содержащую значение свойства message. Поэтому не следует ожидать, что метод toString() будет преобразовывать объект Error в осмысленную строку, понятную человеку. Чтобы выдать пользователю сообщение об ошибке, необходимо явно использовать свойства name и message объекта Error.

Пример

Предупредить об исключении можно посредством следующего кода:

```
function factorial(x) {
    if (x < 0) throw new Error("factorial: x должно быть >= 0");
    if (x <= 1) return 1; else return x * factorial(x-1);
}
```

Перехватывая исключение, можно сообщить о нем пользователю с помощью следующего кода (содержащего клиентский метод Window.alert()):

```
try { &*(&/* здесь возбуждается ошибка */ }
catch(e) {
    if (e instanceof Error) {  // Это экземпляр Error или подкласса?
        alert(e.name + ": " + e.message);
    }
}
```

Смотри также

EvalError, RangeError, ReferenceError, SyntaxError, TypeError, URIError

Error.message

JavaScript 1.5; JScript 5.5; ECMAScript v3

сообщение об ошибке

Синтаксис

error.message

Описание

Свойство message объекта Error (или экземпляра любого подкласса Error) предназначено для хранения понятной человеку строки, содержащей подробные сведения о возникшей ошибке или исключении. Если конструктору Error() передан аргумент message, он становится значением свойства message. Если аргумент message передан не был, объект Error наследует для этого свойства значение по умолчанию, определенное реализацией (которое может быть пустой строкой).

Error.name

JavaScript 1.5; JScript 5.5; ECMAScript v3

тип ошибки

Синтаксис

error.name

Описание

Свойство name объекта Error (или экземпляра любого подкласса Error) задает тип произошедшей ошибки или исключения. Все объекты Error наследуют это свойство от

своего конструктора. Значение свойства совпадает с именем конструктора. Другими словами, у объектов SyntaxError свойство `name` равно «SyntaxError», а у объектов EvalError `name` равно «EvalError».

Error.toString()

JavaScript 1.5; JScript 5.5; ECMAScript v3

преобразует объект Error в строку **перекрывает Object.toString()**

Синтаксис

error.`toString()`

Возвращаемое значение

Строка, определенная реализацией. Стандарт ECMAScript ничего не говорит о возвращаемом этим методом значении за исключением того, что оно должно быть строкой. Стоит отметить, что он не требует, чтобы возвращаемая строка содержала имя ошибки или сообщение об ошибке.

escape()

JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

кодирует строку

Синтаксис

`escape(`*s*`)`

Аргументы

s Строка, которая должна быть закодирована (с помощью управляющих последовательностей).

Возвращаемое значение

Закодированная копия *s*, в которой определенные символы заменены шестнадцатеричными управляющими последовательностями.

Описание

`escape()` – глобальная функция. Она возвращает новую строку, содержащую закодированную версию *s*. Сама строка *s* не изменятся.

Функция `escape()` возвращает строку, в которой все символы *s*, отличные от букв, цифр ASCII и символов пунктуации (@, *, _, +, –, . и /) заменены управляющими последовательностями в формате `%`*xx* или `%u`*xxxx* (где *x* обозначает шестнадцатеричную цифру). Символы Unicode от `\u0000` до `\u00ff` заменяются управляющей последовательностью `%`*xx*, а все остальные символы Unicode – последовательностью `%u`*xxxx*.

Строка, закодированная с помощью `escape()`, декодируется функцией `unescape()`.

В клиентском JavaScript `escape()` чаще всего применяется для кодирования значений cookie, которые имеют ограничения на содержащиеся в них символы пунктуации. См. статью «Document.cookie» в «Справочнике по клиентскому JavaScript».

Хотя функция `escape()` была стандартизована в первой версии ECMAScript, она была признана устаревшей и удалена из стандарта в ECMAScript v3. Реализации ECMAScript скорее всего поддерживают эту функцию, но это не обязательно. В версии JavaScript 1.5, а также в JScript 5.5 и в более поздних вместо `escape()` необходимо применять функции `encodeURI()` и `encodeURIComponent()`.

Пример

```
escape("Hello World!");  // Возвращает "Hello%20World%21"
```

Смотри также

encodeURI(), encodeURIComponent(), String, unescape(); Document.cookie в «Справочнике по клиенсткому JavaScript»

eval() JavaScript 1.0; JScript 1.0; ECMAScript v1

выполняет содержащийся в строке код JavaScript

Синтаксис

eval(*код*)

Аргументы

код

Строка, содержащая выражение JavaScript, которое должно быть вычислено, или инструкции, которые должны быть выполнены.

Возвращаемое значение

Значение выполненного *кода*, если оно есть.

Исключения

SyntaxError

Указывает, что аргумент *код* не содержит корректного кода JavaScript.

EvalError

Указывает, что функция eval() была вызвана некорректно, например через идентификатор, отличный от «eval». См. ограничения, налагаемые на эту функцию, описанные ниже.

Другое исключение

Если код JavaScript, переданный в eval(), генерирует исключение, eval() передаст его вызывающей стороне.

Описание

eval() — это глобальный метод, вычисляющий строку, в которой содержится код JavaScript. Если *код* содержит выражение, eval вычисляет выражение и возвращает его значение. Если *код* содержит одну или несколько инструкций JavaScript, eval() исполняет эти инструкции и возвращает значение (если оно есть), которое возвращает последняя инструкция. Если *код* не возвращает никакого значения, eval() возвращает undefined. И наконец, если *код* генерирует исключение, eval() передает это исключение вызывающей стороне.

eval() предоставляет очень мощные возможности для языка JavaScript, но этот метод не часто используется в реальных программах. Очевидной областью его применения является написание программ, работающих как рекурсивные интерпретаторы JavaScript, или программ, динамически генерирующих и выполняющих код JavaScript.

Большинство функций и методов JavaScript, принимающих строковые аргументы, также принимают аргументы других типов и перед обработкой просто преобразуют эти значения аргументов в строки. eval() ведет себя по-другому. Если аргумент *код*

не является элементарным строковым значением, он возвращается в неизменном виде. Поэтому будьте внимательны, чтобы случайно не передать eval() объект String вместо элементарного строкового значения.

Ради эффективности реализации стандарт ECMAScript v3 налагает на применение eval() необычное ограничение. Реализация ECMAScript разрешает генерировать исключение EvalError, если вы пытаетесь переписать свойство eval или присваиваете метод eval() другому свойству и пытаетесь вызвать его через это свойство.

Пример

```
eval("1+2");  // Возвращает 3
// Этот код использует методы клиентского JavaScript для запроса выражения
// от пользователя и отображения результатов его вычисления.
// Подробности см. в описаниях клиентских методов Window.alert()
// и Window.prompt().
try {
    alert("Результат: " +  eval(prompt("Введите выражение:",""")));
}
catch(exception) {
    alert(exception);
}
var myeval = eval;  // Может генерировать исключение EvalError
myeval("1+2");      // Может генерировать исключение EvalError
```

EvalError

JavaScript 1.5; JScript 5.5; ECMAScript v3

генерируется, когда eval() используется некорректно **наследуется от Error**

Конструктор

new EvalError()

new EvalError(*сообщение*)

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он принимается в качестве значения свойства message объекта EvalError.

Возвращаемое значение

Только что созданный объект EvalError. Если задан аргумент *сообщение*, объект Error возьмет его в качестве значения своего свойства message; в противном случае в качестве значения этого свойства будет использоваться строка по умолчанию, определенная реализацией. Когда конструктор EvalError() вызывается в качестве функции, без оператора new, он ведет себя точно так же, как при вызове с оператором new.

Свойства

message

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. В этом свойстве хранится строка, переданная конструктору, или строка по умолчанию, определенная реализацией. Подробности см. в статье «Error.message».

name

Строка, определяющая тип исключения. Для всех объектов EvalError значение этого свойства равно «EvalError».

Описание

Экземпляры класса EvalError могут создаваться, когда глобальная функция `eval()` вызывается с любым другим именем. Ограничения на способы вызова функции `eval()` рассмотрены в ее описании. Детали о генерировании и перехвате исключений см. в статье «Error».

Смотри также

Error, Error.message, Error.name

Function

JavaScript 1.0; JScript 1.0; ECMAScript v1

функция JavaScript **наследуется от Object**

Синтаксис

```
function имя_функции(список_имен_аргументов)      // Инструкция для определения функции
{
    тело_функции
}
function (список_имен_аргументов) { тело }         // Неименованный функциональный
                                                   // литерал; JavaScript 1.2
имя_функции(список_значений_аргументов)            // Вызов функции
```

Конструктор

```
new Function(имена_аргументов..., тело)          // JavaScript 1.1 и более поздние
```

Аргументы

имена_аргументов...

Любое количество строковых аргументов, которые присваивают имя одному или нескольким аргументам создаваемого объекта Function.

тело

Строка, задающая тело функции. Она может содержать любое количество инструкций JavaScript, разделенных точками с запятой, и может ссылаться на любые имена аргументов, заданные ранее в конструкторе.

Возвращаемое значение

Только что созданный объект Function. Вызов функции приводит к выполнению кода JavaScript, задаваемого *телом*.

Исключения

SyntaxError

Указывает, что в аргументе *тело* или в одном из аргументов в *именах_аргументов* имеется синтаксическая ошибка JavaScript.

Свойства

`arguments[]`

Массив аргументов, переданных функции. Признано устаревшим.

`caller`

Ссылка на объект Function, вызвавший данную функцию, или `null`, если функция была вызвана из кода верхнего уровня. Признано устаревшим.

`length`

Число именованных аргументов, указанных при объявлении функции.

`prototype`

Объект, определяющий для функции конструктора свойства и методы, совместно используемые всеми объектами, созданными с помощью этого конструктора.

Методы

`apply()`

Вызывает функцию как метод заданного объекта, передавая ей указанный массив аргументов.

`call()`

Вызывает функцию как метод заданного объекта, передавая ей аргументы.

`toString()`

Возвращает строковое представление функции.

Описание

Функция – это фундаментальный тип данных JavaScript. В главе 7 рассказано, как определять и использовать функции, а в главе 8 рассматриваются связанные с этим темы о методах, конструкторах и свойствах прототипов функций. Подробности см. в этих главах. Обратите внимание, что функциональные объекты могут создаваться с помощью описанного здесь конструктора `Function()`, но это не эффективно, и в большинстве случаев предпочтительным способом определения функции является инструкция определения функции или функциональный литерал.

В JavaScript 1.1 и более поздних версиях тело функции автоматически получает локальную переменную по имени *arguments*, которая ссылается на объект Arguments. Этот объект представляет собой массив значений, переданных в качестве аргументов функции. Не путайте его с устаревшим свойством `arguments[]`, приведенным выше. Подробности см. в статье «Arguments».

Смотри также

Arguments; главу 7; главу 8

Function.apply()

JavaScript 1.2; JScript 5.5; ECMAScript v3

вызывает функцию как метод объекта

Синтаксис

функция`.apply(`*этот_объект*`, `*аргументы*`)`

Аргументы

этот_объект

Объект, к которому должна быть применена *функция*. В теле функции *этот_объект* становится значением ключевого слова *this*.

аргументы

Массив значений, которые должны быть переданы в качестве аргументов *функции*.

Возвращаемое значение

Значение, возвращаемое при вызове *функции*.

Исключения

TypeError

Если этот метод вызывается для объекта, не являющегося функцией, или если этот метод вызывается с аргументом *аргументы*, не являющимся массивом или объектом Arguments.

Описание

apply() вызывает указанную *функцию*, как если бы она была методом *этого_объекта*, передавая ей аргументы, которые содержатся в массиве *аргументы*. Метод возвращает значение, возвращаемое при вызове функции. В теле функции ключевое слово *this* ссылается на объект *этот_объект*.

Аргумент *аргументы* должен быть массивом или объектом Arguments. Для индивидуального задания передаваемых аргументов, а не в виде элементов массива, применяется Function.call().

Пример

```
// Применяет метод Object.toString() по умолчанию для объекта,
// перекрывающего его собственной версией метода. Обратите внимание на
// отсутствие аргументов.
Object.prototype.toString.apply(o);

// Вызывает метод Math.max(), применяемый для нахождения максимального
// элемента в массиве. Обратите внимание, что в этом случае первый
// аргумент не имеет значения.
var data = [1,2,3,4,5,6,7,8];
Math.max.apply(null, data);
```

Смотри также

Function.call()

Function.arguments[] JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

аргументы, переданные функции

Синтаксис

функция.arguments[*i*]

функция.arguments.length

Описание

Свойство `arguments` объекта Function представляет собой массив аргументов, которые переданы функции. Он определен только во время выполнения функции. `arguments.length` указывает количество элементов в массиве.

Этот свойство признано устаревшим, и вместо него рекомендуется использовать объект Arguments. Несмотря на то что ECMAScript v1 поддерживает свойство `Function.arguments`, оно было убрано из ECMAScript v3, и совместимые реализации могут больше не поддерживать это свойство. Следовательно, его никогда не следует помещать в новый код JavaScript.

Смотри также

Arguments

Function.call() JavaScript 1.5; JScript 5.5; ECMAScript v3

вызывает функцию как метод объекта

Синтаксис

функция`.call(`*этот_объект*, *аргументы...*`)`

Аргументы

этот_объект

Объект, для которого должна быть вызвана *функция*. В теле функции *этот_объект* становится значением ключевого слова `this`.

аргументы...

Любое количество аргументов, которые будут переданы *функции*.

Возвращаемое значение

Значение, возвращаемое при вызове *функции*.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося функцией.

Описание

`call()` вызывает указанную *функцию*, как если бы она была методом *этого_объекта*, передавая ей любые аргументы, расположенные в списке аргументов после *этого_объекта*. `call()` возвращает то, что возвращает вызываемая функция. Внутри тела функции ключевое слово `this` ссылается на объект *этот_объект*.

Если требуется задать аргументы для передачи в функцию в виде массива, используйте `Function.apply()`.

Пример

```
// Вызывает метод Object.toString() по умолчанию для объекта,
// перекрывающего его своей собственной версией метода. Обратите внимание
// на отсутствие аргументов.
Object.prototype.toString.call(o);
```

Смотри также

Function.apply()

Function.caller

JavaScript 1.0, JScript 2.0; устарело в ECMAScript

функция, вызвавшая данную

Синтаксис

функция.caller

Описание

В ранних версиях JavaScript свойство caller объекта Function представляет собой ссылку на функцию, вызвавшую текущую функцию. Если функция была вызвана из верхнего уровня программы JavaScript, caller равно null. Это свойство может использоваться только внутри функции (то есть свойство caller определено для функции, только пока она исполняется).

Свойство Function.caller не является частью стандарта ECMAScript и не обязательно для совместимых реализаций. Его не следует использовать.

Function.length

JavaScript 1.1; JScript 2.0; ECMAScript v1

количество аргументов в объявлении функции

Синтаксис

функция.length

Описание

Свойство length функции указывает количество именованных аргументов, объявленных при определении функции. Функция может быть фактически вызвана с большим или меньшим количеством аргументов. Не путайте это свойство объекта Function со свойством length объекта Arguments, указывающим количество аргументов, фактически переданных функции. Пример см. в статье «Arguments.length».

Смотри также

Arguments.length

Function.prototype

JavaScript 1.1; JScript 2.0; ECMAScript v1

прототип для класса объектов

Синтаксис

функция.prototype

Описание

Свойство prototype применяется, когда функция вызывается как конструктор. Оно ссылается на объект, который служит в качестве прототипа для целого класса объектов. Любой объект, созданный с помощью конструктора, наследует все свойства объекта, на который ссылается свойство prototype.

Обсуждение функций-конструкторов, свойства prototype и определение классов в JavaScript см. в главе 8.

Ошибки

JavaScript 1.1 требует, чтобы конструктор был вызван до того, как что-нибудь сможет быть присвоено его объекту-прототипу.

Смотри также

Главу 8

Function.toString()

JavaScript 1.0; JScript 2.0; ECMAScript v1

преобразует функцию в строку

Синтаксис

функция.toString()

Возвращаемое значение

Строка, представляющая функцию.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося функцией.

Описание

Метод toString() объекта Function преобразует функцию в строку способом, зависящим от реализации. В реализациях Netscape этот метод возвращает строку корректного кода JavaScript, включающего ключевое слово function, список аргументов, полное тело функции и т. д.

Global

JavaScript 1.0; JScript 1.0; ECMAScript v1

глобальный объект

Синтаксис

this

Глобальные свойства

Глобальный объект – это не класс, поэтому для следующих глобальных свойств имеются самостоятельные справочные статьи под собственными именами. То есть подробные сведения о свойстве undefined можно найти под заголовком «undefined», а не «Global.undefined». Обратите внимание, что все переменные верхнего уровня также представляют собой свойства глобального объекта.

Infinity

Числовое значение, обозначающее положительную бесконечность.

NaN

Нечисловое значение.

undefined

Значение undefined.

Глобальные функции

Глобальный объект – это объект, а не класс. Глобальные функции, перечисленные ниже, не являются методами какого-либо объекта, и справочные статьи приведены под именами функций. Так, функция parseInt() подробно описывается под заголовком «parseInt()», а не «Global.parseInt()».

`decodeURI()`

Декодирует строку, закодированную функцией encodeURI().

`decodeURIComponent()`

Декодирует строку, закодированную с помощью encodeURIComponent().

`encodeURI`

Кодирует URI, заменяя определенные символы управляющими последовательностями.

`encodeURIComponent`

Кодирует компонент URI, заменяя определенные символы управляющими последовательностями.

`escape()`

Кодирует строку, заменяя определенные символы управляющими последовательностями.

`eval()`

Вычисляет строку кода JavaScript и возвращает результат.

`isFinite()`

Проверяет, является ли значение конечным числом.

`isNaN`

Проверяет, является ли значение нечисловым значением (NaN).

`parseFloat()`

Выбирает число из строки.

`parseInt()`

Выбирает целое из строки.

`unescape()`

Декодирует строку, закодированную с помощью escape().

Глобальные объекты

В дополнение к глобальным свойствам и функциям, перечисленным выше, глобальный объект также определяет свойства, ссылающиеся на все остальные предопределенные объекты JavaScript. Все эти свойства являются функциями-конструкторами, определяющими классы, за исключением `Math`, которое представляет собой ссылку на объект, не являющийся конструктором.

`Array`

Конструктор `Array()`.

`Boolean`

Конструктор `Boolean()`.

`Date`

Конструктор `Date()`.

`Error`

Конструктор `Error()`.

`EvalError`

Конструктор `EvalError()`.

`Function`

Конструктор `Function()`.

`Math`

Ссылка на объект, определяющий математические функции.

`Number`

Конструктор `Number()`.

`Object`

Конструктор `Object()`.

`RangeError`

Конструктор `RangeError()`.

`ReferenceError`

Конструктор `ReferenceError()`.

`RegExp`

Конструктор `RegExp()`.

`String`

Конструктор `String()`.

`SyntaxError`

Конструктор `SyntaxError()`.

`TypeError`

Конструктор `TypeError()`.

`URIError`

Конструктор `URIError()`.

Описание

Глобальный объект – это предопределенный объект, который служит для размещения глобальных свойств и функций JavaScript. Все остальные предопределенные объекты, функции и свойства доступны через глобальный объект. Глобальный объект не является свойством любого другого объекта, поэтому у него нет имени. (Заголовок справочной статьи выбран просто для удобства организации и не указывает на то, что глобальный объект имеет имя «Global».) В коде JavaScript верхнего уровня можно ссылаться на глобальный объект посредством ключевого слова `this`. Однако этот способ обращения к глобальному объекту требуется редко, так как глобальный объект выступает в качестве начала цепочки областей видимости, и поэтому поиск неуточненных имен переменных и функций выполняется среди свойств этого объекта. Когда код JavaScript ссылается, например, на функцию `parseInt()`, он ссылается на свойство `parseInt` глобального объекта. Тот факт, что глобальный объект находится в начале цепочки областей видимости, также означает, что все переменные, объявленные в коде JavaScript верхнего уровня, становятся свойствами глобального объекта.

Глобальный объект – это просто объект, а не класс. Нет конструктора `Global()`, и нет способа создать экземпляр нового глобального объекта.

Когда JavaScript встраивается в определенную среду, глобальному объекту обычно даются дополнительные свойства, специфические для этой среды. На самом деле тип глобального объекта в стандарте ECMAScript не указан, и в конкретной реализации

JavaScript в качестве глобального может выступать объект любого типа, если только объект определяет основные свойства и функции, перечисленные здесь. Например, в клиентском JavaScript глобальным объектом является объект Window, представляющий окно веб-броузера, внутри которого исполняется код JavaScript.

Пример

В базовом JavaScript ни одно из предопределенных свойств не является перечисляемым, благодаря чему можно получить список всех явно и неявно объявленных глобальных переменных с помощью следующего цикла `for/in`:

```
var variables = ""
for(var name in this)
    variables += name + "\n";
```

Смотри также

Window в «Справочнике по клиентскому JavaScript»; главу 4

Infinity

JavaScript 1.3; JScript 3.0; ECMAScript v1

числовое свойство, обозначающее бесконечность

Синтаксис

`Infinity`

Описание

`Infinity` — это глобальное свойство, содержащее специальное числовое значение, которое обозначает положительную бесконечность. Свойство `Infinity` не перечисляется циклами `for/in` и не может быть удалено с помощью оператора `delete`. Заметьте, что `Infinity` не является константой и может быть установлено равным какому-либо другому значению, но лучше этого не делать. (Однако `Number.POSITIVE_INFINITY` — это константа.)

Смотри также

isFinite(), NaN, Number.POSITIVE_INFINITY

isFinite()

JavaScript 1.2; JScript 3.0; ECMAScript v1

определяет, является ли число конечным

Синтаксис

`isFinite(`*n*`)`

Аргументы

n — проверяемое число.

Возвращаемое значение

`true`, если *n* является конечным числом (или может быть преобразовано в него), или `false`, если *n* — это `NaN` (нечисло) или плюс/минус бесконечность.

Смотри также

Infinity, isNaN(), NaN, Number.NaN, Number.NEGATIVE_INFINITY, Number. POSITIVE_INFINITY

isNaN()

JavaScript 1.1; JScript 1.0; ECMAScript v1

проверка на нечисло

Синтаксис

isNaN(*x*)

Аргументы

x – проверяемое значение.

Возвращаемое значение

true, если *x* является специальным нечисловым значением (или может быть в него преобразовано); false, если *x* – это любое другое значение.

Описание

isNaN() проверяет свой аргумент, чтобы определить, не представляет ли собой его значение NaN, то есть некорректное число (например, результат деления на ноль). Эта функция нужна, так как сравнение NaN с любым значением, включая само себя, всегда возвращает false, поэтому нельзя проверить на NaN с помощью операторов == или ===.

Обычное применение функции isNaN() – проверка результатов parseFloat() и parseInt(), преследующая цель определить, представляют ли они корректные числа. Функция isNaN() также применяется для проверки на арифметические ошибки, такие как деление на ноль.

Пример

```
isNaN(0);                  // Возвращает false
isNaN(0/0);                // Возвращает true
isNaN(parseInt("3"));      // Возвращает false
isNaN(parseInt("hello"));  // Возвращает true
isNaN("3");                // Возвращает false
isNaN("hello");            // Возвращает true
isNaN(true);               // Возвращает false
isNaN(undefined);          // Возвращает true
```

Смотри также

isFinite(), NaN, Number.NaN, parseFloat(), parseInt()

Math

JavaScript 1.0; JScript 1.0; ECMAScript v1

математические функции и константы

Синтаксис

Math.*константа*

Math.*функция*()

Константы

Math.E

Константа *e*, основание натуральных логарифмов.

Math.LN10

Натуральный логарифм 10.

`Math.LN2`

Натуральный логарифм 2.

`Math.LOG10E`

Десятичный логарифм *e*.

`Math.LOG2E`

Логарифм *e* по основанию 2.

`Math.PI`

Константа π.

`Math.SQRT1_2`

Единица, деленная на корень квадратный из 2.

`Math.SQRT2`

Квадратный корень из 2.

Статические функции

`Math.abs()`

Вычисляет абсолютное значение.

`Math.acos()`

Вычисляет арккосинус.

`Math.asin()`

Вычисляет арксинус.

`Math.atan()`

Вычисляет арктангенс.

`Math.atan2()`

Вычисляет угол между осью X и точкой.

`Math.ceil()`

Округляет число вверх.

`Math.cos()`

Вычисляет косинус.

`Math.exp()`

Вычисляет степень числа *e*.

`Math.floor()`

Округляет число вниз.

`Math.log()`

Вычисляет натуральный логарифм.

`Math.max()`

Возвращает большее из двух чисел.

`Math.min()`

Возвращает меньшее из двух чисел.

`Math.pow()`

Вычисляет x^y.

`Math.random()`

Возвращает случайное число.

`Math.round()`

Округляет до ближайшего целого.

`Math.sin()`

Вычисляет синус.

`Math.sqrt()`

Вычисляет квадратный корень.

`Math.tan()`

Вычисляет тангенс.

Описание

Math – это объект, определяющий свойства, которые ссылаются на полезные функции и константы. Эти функции и константы удобно объединены объектом Math и вызываются с помощью следующего синтаксиса:

```
y = Math.sin(x);
area = radius * radius * Math.PI;
```

Math – это не класс объектов, как Date и String. Конструктора `Math()` нет, и такие функции, как `Math.sin()`, – это просто функции, а не методы, работающие с объектом.

Смотри также

Number

Math.abs() JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет абсолютное значение

Синтаксис

`Math.abs(x)`

Аргументы

x – любое число.

Возвращаемое значение

Абсолютное значение *x*.

Math.acos() JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет арккосинус

Синтаксис

`Math.acos(x)`

Аргументы

x – число между –1.0 и 1.0.

Возвращаемое значение

Арккосинус указанного числа *x*. Возвращаемое значение может находиться в интервале от 0 до π радиан.

Math.asin()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет арксинус

Синтаксис

`Math.asin(x)`

Аргументы

x – число между –1.0 и 1.0.

Возвращаемое значение

Арксинус указанного значения *x*. Это возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет арктангенс

Синтаксис

`Math.atan(x)`

Аргументы

x – любое число.

Возвращаемое значение

Арктангенс указанного значения *x*. Возвращаемое значение может находиться в интервале от $-\pi/2$ до $\pi/2$ радиан.

Math.atan2()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет угол между осью X и точкой

Синтаксис

`Math.atan2(y, x)`

Аргументы

y – координата Y точки.

x – координата X точки.

Возвращаемое значение

Значение между $-\pi$ и π радиан, указывающее на угол по направлению, обратному часовой стрелке, между положительной осью X и точкой (*x*, *y*).

Описание

Функция `Math.atan2()` вычисляет арктангенс отношения *y*/*x*. Аргумент *y* может рассматриваться как координата Y (или «рост») точки, а аргумент *x* – как координата X (или «пробег») точки. Обратите внимание на необычный порядок аргументов этой функции: координата Y передается до координаты X.

Math.ceil()

JavaScript 1.0; JScript 1.0; ECMAScript v1

округляет число

Синтаксис

`Math.ceil(x)`

Аргументы

x – числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, большее или равное *x*.

Описание

`Math.ceil()` вычисляет функцию ceiling, то есть возвращает ближайшее целое, большее или равное аргументу функции. `Math.ceil()` отличается от `Math.round()` тем, что округляет всегда вверх, а не к ближайшему целому. Обратите внимание также, что `Math.ceil()` не округляет отрицательные числа к большим по абсолютному значению отрицательным целым; функция округляет их вверх по направлению к нулю.

Пример

```
a = Math.ceil(1.99);   // Результат равен 2.0
b = Math.ceil(1.01);   // Результат равен 2.0
c = Math.ceil(1.0);    // Результат равен 1.0
d = Math.ceil(-1.99);  // Результат равен -1.0
```

Math.cos()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет косинус

Синтаксис

`Math.cos(x)`

Аргументы

x – угол, измеренный в радианах. Чтобы преобразовать градусы в радианы, умножьте значение в градусах на 0,017453293 ($2\pi/360$).

Возвращаемое значение

Косинус указанного значения *x*. Это возвращаемое значение может находиться в интервале между –1.0 и 1.0.

Math.E

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа *e*

Синтаксис

`Math.E`

Описание

`Math.E` – это математическая константа *e*, база натуральных логарифмов, приблизительно равная 2,71828.

Math.exp()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет e^x

Синтаксис

`Math.exp(x)`

Аргументы

x – число или выражение, которое должно использоваться как экспонента.

Возвращаемое значение

e^x – это e, возведенное в степень указанной экспоненты x, где e – это основание натуральных логарифмов, примерно равное 2,71828.

Math.floor()

JavaScript 1.0; JScript 1.0; ECMAScript v1

округляет число вниз

Синтаксис

`Math.floor(x)`

Аргументы

x – числовое значение или выражение.

Возвращаемое значение

Ближайшее целое, меньшее или равное x.

Описание

`Math.floor()` вычисляет функцию floor – другими словами, она возвращает ближайшее целое значение, меньшее или равное аргументу функции.

`Math.floor()` округляет вещественное число вниз – в отличие от `Math.round()`, округляющей до ближайшего целого. Также заметьте, что `Math.floor()` округляет отрицательные числа вниз (то есть дальше от нуля), а не вверх (то есть ближе к нулю).

Пример

```
a = Math.floor(1.99);    // Результат равен 1.0
b = Math.floor(1.01);    // Результат равен 1.0
c = Math.floor(1.0);     // Результат равен 1.0
d = Math.floor(-1.01);   // Результат равен -2.0
```

Math.LN10

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $\log_e 10$

Синтаксис

`Math.LN10`

Описание

`Math.LN10` – это $\log_e 10$, натуральный логарифм 10. Эта константа имеет значение, приблизительно равное 2,3025850929940459011.

Math.LN2

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $\log_e 2$

Синтаксис

```
Math.LN2
```

Описание

`Math.LN2` – это $\log_e 2$, натуральный логарифм 2. Эта константа имеет значение, приблизительно равное 0,69314718055994528623.

Math.log()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет натуральный логарифм

Синтаксис

```
Math.log(x)
```

Аргументы

x – любое числовое значение, большее или равное нулю.

Возвращаемое значение

Натуральный логарифм *x*.

Описание

`Math.log()` вычисляет натуральный логарифм своего аргумента. Аргумент должен быть больше нуля.

Логарифмы числа по основанию 10 и по основанию 2 можно вычислить по следующим формулам:

$$\log_{10} x = \log_{10} e \times \log_e x$$

$$\log_2 x = \log_2 e \times \log_e x$$

Эти формулы транслируются в следующие функции JavaScript:

```
function log10(x) { return Math.LOG10E * Math.log(x); }
function log2(x) { return  Math.LOG2E * Math.log(x); }
```

Math.LOG10E

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $\log_{10} e$

Синтаксис

```
Math.LOG10E
```

Описание

`Math.LOG10E` – это $\log_{10} e$, логарифм по основанию 10 константы e. Его значение приблизительно равно 0,43429448190325181667.

Math.LOG2E

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $\log_2 e$

Синтаксис

`Math.LOG2E`

Описание

`Math.LOG2E` — это $\log_2 e$, логарифм по основанию 2 константы e. Его значение приблизительно равно 1,442695040888963387.

Math.max()

JavaScript 1.0; JScript 1.0; ECMAScript v1; расширена в ECMAScript v3

возвращает наибольший аргумент

Синтаксис

`Math.max(`*аргументы...*`)`

Аргументы

аргументы...

Ноль или более значений. До ECMAScript v3 этот метод мог принимать ровно два аргумента.

Возвращаемое значение

Наибольший из аргументов. Возвращает `-Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов равен `NaN` или является нечисловым значением, которое не может быть преобразовано в число.

Math.min()

JavaScript 1.0; JScript 1.0; ECMAScript v1; расширена в ECMAScript v3

возвращает наименьший аргумент

Синтаксис

`Math.min(`*аргументы...*`)`

Аргументы

аргументы...

Любое количество аргументов. До ECMAScript v3 эта функция принимала ровно два аргумента.

Возвращаемое значение

Наименьший из указанных аргументов. Возвращает `Infinity`, если аргументов нет. Возвращает `NaN`, если какой-либо из аргументов представляет собой `NaN` или нечисловое значение и не может быть преобразован в число.

Math.PI

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа π

Синтаксис

`Math.PI`

Описание

`Math.PI` — это константа π, или «пи», отношение длины окружности к ее диаметру. Имеет значение, примерно равное 3,14159265358979.

Math.pow()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет x^y

Синтаксис

`Math.pow(x, y)`

Аргументы

x — число, которое должно быть возведено в степень.

y — степень, в которую должно быть возведено *x*.

Возвращаемое значение

x^y, то есть *x* в степени *y*.

Описание

`Math.pow()` вычисляет *x* в степени *y*. Значения *x* и *y* могут быть любыми. Однако если результат является мнимым или комплексным числом, `Math.pow()` возвращает `NaN`. На практике это означает, что если *x* отрицательно, то *y* должно быть положительным или отрицательным целым. Также имейте в виду, что большие экспоненты легко приводят к вещественному переполнению и возвращают значение `Infinity`.

Math.random()

JavaScript 1.1; JScript 1.0; ECMAScript v1

возвращает псевдослучайное число

Синтаксис

`Math.random()`

Возвращаемое значение

Псевдослучайное число между 0.0 и 1.0.

Math.round()

JavaScript 1.0; JScript 1.0; ECMAScript v1

округляет до ближайшего целого

Синтаксис

`Math.round(x)`

Аргументы

x — любое число.

Возвращаемое значение

Целое, ближайшее к *x*.

Описание

`Math.round()` округляет аргумент вверх или вниз до ближайшего целого. 1/2 округляется вверх. Например, 2,5 округляется до 3, а –2,5 округляется до –2.

Math.sin()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет синус

Синтаксис

`Math.sin(x)`

Аргументы

x – угол в радианах. Для преобразования градусов в радианы умножьте число на 0,017453293 ($2\pi/360$).

Возвращаемое значение

Синус *x*.

Math.sqrt()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет квадратный корень

Синтаксис

`Math.sqrt(x)`

Аргументы

x – числовое значение, большее или равное 0.

Возвращаемое значение

Квадратный корень из *x*. Возвращает `NaN`, если *x* меньше нуля.

Описание

`Math.sqrt()` вычисляет квадратный корень из числа. Обратите внимание, однако, что произвольные корни из чисел можно вычислять посредством `Math.pow()`. Например:

```
Math.cuberoot = function(x){ return Math.pow(x,1/3); }
Math.cuberoot(8);  // Возвращает 2
```

Math.SQRT1_2

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $1/\sqrt{2}$

Синтаксис

`Math.SQRT1_2`

Описание

`Math.SQRT1_2` – это $1/\sqrt{2}$, обратная величина корня квадратного из 2. Эта константа примерно равна 0,7071067811865476.

Math.SQRT2

JavaScript 1.0; JScript 1.0; ECMAScript v1

математическая константа $\sqrt{2}$

Синтаксис

`Math.SQRT2`

Описание

`Math.SQRT2` — это $\sqrt{2}$, корень квадратный из 2. Эта константа имеет значение, примерно равное 1,414213562373095.

Math.tan()

JavaScript 1.0; JScript 1.0; ECMAScript v1

вычисляет тангенс

Синтаксис

`Math.tan(x)`

Аргументы

x — угол, измеряемый в радианах. Чтобы преобразовать градусы в радианы, умножьте значение в градусах на 0,017453293 ($2\pi/360$).

Возвращаемое значение

Тангенс указанного угла *x*.

NaN

JavaScript 1.3; JScript 3.0; ECMAScript v1

свойство «нечисло»

Синтаксис

`NaN`

Описание

`NaN` — это глобальное свойство, ссылающееся на специальное числовое значение «нечисло». Свойство `NaN` не перечисляется циклами `for/in` и не может быть удалено оператором `delete`. Обратите внимание, что `NaN` — это не константа, и оно может быть установлено в любое значение, но лучше этого не делать.

Определить, является ли значение нечислом, можно посредством `isNaN()`, так как `NaN` всегда при сравнении оказывается неравным любой другой величине, включая само себя!

Смотри также

Infinity, isNaN(), Number.NaN

Number

JavaScript 1.1; JScript 2.0; ECMAScript v1

поддержка чисел **наследуется от Object**

Конструктор

`new Number(значение)`

`Number(значение)`

Аргументы

значение

Числовое значение создаваемого объекта Number или значение, которое может быть преобразовано в число.

Возвращаемое значение

Когда `Number()` используется с оператором `new` в качестве конструктора, он возвращает только что созданный объект Number. Когда `Number()` вызывается как функция без оператора `new`, она преобразует свой аргумент в элементарное числовое значение и возвращает это значение (или `NaN`, если преобразование не удалось).

Константы

`Number.MAX_VALUE`

Наибольшее представимое число.

`Number.MIN_VALUE`

Наименьшее представимое число.

`Number.NaN`

Значение «нечисло».

`Number.NEGATIVE_INFINITY`

Значение «отрицательная бесконечность», возвращается в случае переполнения.

`Number.POSITIVE_INFINITY`

Положительная бесконечность; возвращается при переполнении.

Методы

`toString()`

Преобразует число в строку в указанной системе счисления.

`toLocaleString()`

Преобразует число в строку, руководствуясь локальными соглашениями о форматировании чисел.

`toFixed()`

Преобразует число в строку, содержащую указанное число цифр после десятичной точки.

`toExponential()`

Преобразует числа в строки в экспоненциальной нотации с указанным количеством цифр после десятичной точки.

`toPrecision()`

Преобразует число в строку, записывая в нее указанное количество значащих цифр. Нотация экспоненциальная или с фиксированной точкой в зависимости от размера числа и указанного количества значащих цифр.

Описание

Числа – это базовый, элементарный тип данных в JavaScript. Однако в JavaScript 1.1 также поддерживается объект Number, представляющий собой обертку вокруг элементарного числового значения. JavaScript при необходимости автоматически выполняет преобразование между элементарной и объектной формой. В JavaScript 1.1 можно явно создать объект Number посредством конструктора `Number()`, хотя в этом редко возникает необходимость.

Конструктор `Number()` может также вызываться без оператора `new`, как функция преобразования. В этом случае он пытается преобразовать свой аргумент в число и возвращает элементарное числовое значение (или NaN), полученное при преобразовании.

Конструктор Number() также применяется для размещения пяти полезных числовых констант: максимальное и минимальное представимые числа, положительная и отрицательная бесконечность и специальное значение «нечисло». Обратите внимание, что эти значения представляют собой свойства самой функции-конструктора Number(), а не индивидуальных числовых объектов. Например, можно применять свойство MAX_VALUE следующим образом:

```
var biggest = Number.MAX_VALUE
```

но *не* так:

```
var n = new Number(2);
var biggest = n.MAX_VALUE
```

Но toString() и другие методы объекта Number – это методы каждого объекта Number, а не функции-конструктора Number(). Как уже говорилось, JavaScript по необходимости автоматически выполняет преобразования между элементарными числовыми значениями и объектами Number. То есть методы класса Number могут работать с элементарными числовыми значениями так же, как с объектами Number.

```
var value = 1234;
var binary_value = n.toString(2);
```

Смотри также

Infinity, Math, NaN

Number.MAX_VALUE

JavaScript 1.1; JScript 2.0, ECMAScript v1

максимальное числовое значение

Синтаксис

Number.MAX_VALUE

Описание

Number.MAX_VALUE – это наибольшее число, представимое в JavaScript. Его значение примерно равно 1.79E+308.

Number.MIN_VALUE

JavaScript 1.1; JScript 2.0, ECMAScript v1

минимальное числовое значение

Синтаксис

Number.MIN_VALUE

Описание

Number.MIN_VALUE – это наименьшее число (ближайшее к нулю, а не самое отрицательное), представимое в JavaScript. Его значение примерно равно 5E–324.

Number.NaN

JavaScript 1.1; JScript 2.0, ECMAScript v1

специальное нечисловое значение

Синтаксис

`Number.NaN`

Описание

`Number.NaN` — это специальное значение, указывающее, что результат некоторой математической операции (например, извлечения квадратного корня из отрицательного числа) не является числом. `parseInt()` и `parseFloat()` возвращают это значение, когда не могут преобразовать указанную строку в число, и программист может применять `Number.NaN` аналогичным образом, чтобы указать на ошибочное условие для какой-либо функции, обычно возвращающей допустимое число.

JavaScript печатает значение `Number.NaN` как `NaN`. Обратите внимание, что значение `NaN` всегда при сравнении не равно любому другому числу, включая само `NaN`. Следовательно, невозможно проверить значение на «нечисло», сравнив его с `Number.NaN`. Для этого предназначена `isNaN()`. В ECMAScript v1 и более поздних версиях вместо `Number.NaN` допускается использование предопределенной глобальной константы `NaN`.

Смотри также

isNaN(), NaN

Number.NEGATIVE_INFINITY

JavaScript 1.1; JScript 2.0, ECMAScript v1

отрицательная бесконечность

Синтаксис

`Number.NEGATIVE_INFINITY`

Описание

`Number.NEGATIVE_INFINITY` — специальное числовое значение, возвращаемое, если арифметическая операция или математическая функция генерирует отрицательное число, большее чем максимальное представимое в JavaScript число (то есть отрицательное число, меньшее чем `-Number.MAX_VALUE`).

JavaScript отображает значение `NEGATIVE_INFINITY` как `-Infinity`. Это значение математически ведет себя как бесконечность. Например, все, что угодно, умноженное на бесконечность, является бесконечностью, а все, деленное на бесконечность, — это ноль. В ECMAScript v1 и более поздних версиях можно также использовать `-Infinity` вместо `Number.NEGATIVE_INFINITY`.

Смотри также

Infinity, isFinite()

Number.POSITIVE_INFINITY

JavaScript 1.1; JScript 2.0, ECMAScript v1

бесконечность

Синтаксис

`Number.POSITIVE_INFINITY`

Описание

`Number.POSITIVE_INFINITY` — это специальное числовое значение, возвращаемое, когда арифметическая операция или математическая функция приводит к переполнению или генерирует значение, превосходящее максимальное представимое число в JavaScript (то есть `Number.MAX_VALUE`). Обратите внимание, что если происходит потеря значимости или число становится меньше, чем `Number.MIN_VALUE`, JavaScript преобразует его в ноль.

JavaScript отображает значение `POSITIVE_INFINITY` как `Infinity`. Это значение ведет себя математически так же, как бесконечность. Например, что-либо, умноженное на бесконечность – это бесконечность, а что-либо, деленное на бесконечность, равно нулю. В ECMAScript v1 и более поздних версиях вместо `Number.POSITIVE_INFINITY` можно также использовать предопределенную глобальную константу `Infinity`.

Смотри также

Infinity, isFinite()

Number.toExponential()

JavaScript 1.5; JScript 5.5, ECMAScript v3

форматирует число в экспоненциальной нотации

Синтаксис

число`.toExponential(`*цифры*`)`

Аргументы

цифры

Количество цифр после десятичной точки. Может быть значением от 0 до 20 включительно, конкретные реализации могут поддерживать больший диапазон значений. Если аргумент отсутствует, то цифр будет столько, сколько необходимо.

Возвращаемое значение

Строковое представление *числа* в экспоненциальной нотации, с одной цифрой перед десятичной точкой и с количеством цифр, указанным в аргументе *цифры*, после нее. Дробная часть, если это необходимо, округляется или дополняется нулями, чтобы она имела указанную длину.

Исключения

`RangeError`

Если аргумент *цифры* слишком велик или слишком мал. Значения между 0 и 20 включительно не приводят к ошибке RangeError. Реализациям также разрешено поддерживать большее или меньшее количество цифр.

`TypeError`

Если метод вызывается для объекта, не являющегося Number.

Пример

```
var n = 12345.6789;
n.toExponential(1);   // Возвращает 1.2e+4
n.toExponential(5);   // Возвращает 1.23457e+4
n.toExponential(10);  // Возвращает 1.2345678900e+4
n.toExponential();    // Возвращает 1.23456789e+4
```

Смотри также

Number.toFixed(), Number.toLocaleString(), Number.toPrecision(), Number.toString()

Number.toFixed() JavaScript 1.5; JScript 5.5, ECMAScript v3

форматирует число в нотации с фиксированной точкой

Синтаксис

число.toFixed(*цифры*)

Аргументы

цифры

Количество цифр после десятичной точки; оно может быть значением от 0 до 20 включительно; конкретные реализации могут поддерживать больший диапазон значений. Если этот аргумент отсутствует, он считается равным 0.

Возвращаемое значение

Строковое представление *числа*, которое не использует экспоненциальную нотацию и в котором количество цифр после десятичной точки равно аргументу *цифры*. По необходимости число округляется, а дробная часть дополняется нулями до указанной длины. Если *число* больше, чем 1e+21, этот метод вызывает Number.toString() и возвращает строку в экспоненциальной нотации.

Исключения

RangeError

Если аргумент *цифры* слишком велик или слишком мал. Значения от 0 до 20 включительно не приводят к RangeError. Конкретные реализации допускают большие или меньшие значения.

TypeError

Если метод вызывается для объекта, не являющегося Number.

Пример

```
var n = 12345.6789;
n.toFixed();          // Возвращает 12346: обратите внимание на округление
                      // и отсутствие дробной части
n.toFixed(1);          // Возвращает 12345.7: обратите внимание на округление
n.toFixed(6);         // Возвращает 12345.678900: обратите внимание
                      // на добавление нулей
(1.23e+20).toFixed(2);// Возвращает 123000000000000000000.00
(1.23e-10).toFixed(2) // Возвращает 0.00
```

Смотри также

Number.toExponential(), Number.toLocaleString(), Number.toPrecision(), Number. toString()

Number.toLocaleString()

JavaScript 1.5; JScript 5.5, ECMAScript v3

преобразует число в строку, отформатированную в соответствии с локальными соглашениями

Синтаксис

число.toLocaleString()

Возвращаемое значение

Зависящее от реализации строковое представление числа, отформатированное в соответствии с локальными соглашениями, на которое могут влиять, например, символы пунктуации, выступающие в качестве десятичной точки и разделителя тысяч.

Исключения

TypeError

Если метод вызван для объекта, не являющегося Number.

Смотри также

Number.toExponential(), Number.toFixed(), Number.toPrecision(), Number.toString()

Number.toPrecision()

JavaScript 1.5; JScript 5.5, ECMAScript v3

форматирует значащие цифры числа

Синтаксис

число.toPrecision(*точность*)

Аргументы

точность

Количество значащих цифр в возвращаемой строке. Оно может быть значением от 1 до 21 включительно. Конкретные реализации могут поддерживать большие и меньшие значения *точности*. Если этот аргумент отсутствует, для преобразования в десятичное число применяется метод toString().

Возвращаемое значение

Строковое представление *числа*, содержащее количество значащих цифр, определяемое аргументом *точность*. Если *точность* достаточна для включения всех цифр целой части *числа*, возвращаемая строка записывается в нотации с фиксированной точкой. В противном случае запись осуществляется в экспоненциальной нотации с одной цифрой перед десятичной точкой и *точность*–1 цифр после десятичной точки. Число по необходимости округляется или дополняется нулями.

Исключения

RangeError

Если аргумент *цифры* слишком мал или слишком велик. Значения от 1 до 21 включительно не приводят к RangeError. Конкретные реализации могут поддерживать большие и меньшие значения.

TypeError

Если метод вызывается для объекта, не являющегося Number.

Пример

```
var n = 12345.6789;
n.toPrecision(1);   // Возвращает 1e+4
n.toPrecision(3);   // Возвращает 1.23e+4
n.toPrecision(5);   // Возвращает 12346: обратите внимание на округление
n.toPrecision(10);  // Возвращает 12345.67890: обратите внимание
                    // на добавление нуля
```

Смотри также

Number.toExponential(), Number.toFixed(), Number.toLocaleString(), Number.toString()

Number.toString()

JavaScript 1.1; JScript 2.0, ECMAScript v1

преобразует число в строку **перекрывает Object.toString()**

Синтаксис

число.toString(*основание*)

Аргументы

основание

Необязательный аргумент, задающий основание системы счисления, между 2 и 36, в которой должно быть представлено число. Если аргумент отсутствует, то основание равно 10. Заметьте, однако, что спецификация ECMAScript разрешает реализации возвращать любое значение, если этот аргумент равен любому значению, отличному от 10.

Возвращаемое значение

Строковое представление числа.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося Number.

Описание

Метод toString() объекта Number преобразует число в строку. Если аргумент *основание* опущен или указано значение 10, число преобразуется в строку по основанию 10. Если *основание* равно любому другому значению, этот метод возвращает строку, определяемую реализацией. Реализации от Netscape и Microsoft после JScript 3.0 учитывают аргумент *основание* и возвращают строковое представление числа в системе счисления с указанным основанием.

Смотри также

Number.toExponential(), Number.toFixed(), Number.toLocaleString(), Number.toPrecision()

Number.valueOf()

JavaScript 1.1; JScript 2.0, ECMAScript v1

возвращает элементарное числовое значение **перекрывает Object.valueOf()**

Синтаксис

число.valueOf()

Возвращаемое значение

Элементарное числовое значение объекта Number. В явном вызове этого метода редко возникает необходимость.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося Number.

Смотри также

Object.valueOf()

Object

JavaScript 1.0; JScript 1.0; ECMAScript v1

надкласс, реализующий
общие возможности всех объектов JavaScript

Конструктор

new Object()

new Object(*значение*)

Аргументы

значение

В этом необязательном аргументе указано элементарное значение JavaScript – число, логическое значение или строка, – которое должно быть преобразовано в объект Number, Boolean или String. Этот объект не поддерживается в версиях до JavaScript 1.1 и ECMAScript v1.

Возвращаемое значение

Если передан аргумент *значение*, этот конструктор возвращает только что созданный экземпляр Object. Если указан аргумент *значение* элементарного типа, конструктор создает объект-обертку Number, Boolean или String для указанного элементарного значения. Если конструктор Object() вызывается как функция, он ведет себя точно так же, как при вызове без оператора new.

Свойства

constructor

Ссылка на функцию JavaScript, которая была конструктором объекта.

Методы

hasOwnProperty()

Проверяет, имеет ли объект локально определенное (не унаследованное) свойство с указанным именем.

`isPrototypeOf()`

Проверяет, является ли данный объект прототипом для указанного объекта.

`propertyIsEnumerable()`

Проверяет, существует ли свойство с указанным именем и будет ли оно перечислено циклом `for/in`.

`toLocaleString()`

Возвращает локализованное строковое представление объекта. Реализация по умолчанию этого метода просто вызывает `toString()`, но подклассы могут перекрывать его для выполнения локализации.

`toString()`

Возвращает строковое представление объекта. Реализация этого метода в классе Object является очень общей и возвращает не много полезной информации. Подклассы Object обычно перекрывают этот метод, определяя собственный метод `toString()`, выдающий более полезный результат.

`valueOf()`

Возвращает элементарное значение объекта, если оно существует. Для объектов типа Object этот метод просто возвращает сам объект. Подклассы Object, такие как Number и Boolean, перекрывают этот метод, чтобы можно было получить элементарное значение, связанное с объектом.

Описание

Класс Object – это встроенный тип данных языка JavaScript. Он выступает в качестве надкласса для всех остальных объектов JavaScript; следовательно, методы и поведение класса Object наследуются всеми остальными объектами. Базовое поведение объектов в JavaScript разъясняется в главе 8.

В дополнение к конструктору `Object()`, показанному выше, объекты могут также создаваться и инициализироваться с помощью синтаксиса объектных литералов, описанного в главе 8.

Смотри также

Array, Boolean, Function, Function.prototype, Number, String; главу 8

Object.constructor

JavaScript 1.1; JScript 2.0; ECMAScript v1

функция-конструктор объекта

Синтаксис

объект.`constructor`

Описание

Свойство `constructor` любого объекта – это ссылка на функцию, которая выступала в качестве конструктора для этого объекта. Например, если массив `a` создается с помощью конструктора `Array()`, то `a.constructor` будет равно `Array`:

```
a = new Array(1,2,3);   // Создает объект
a.constructor == Array  // Равно true
```

Одно из распространенных применений свойства constructor состоит в определении типа неизвестных объектов. Оператор typeof позволяет определить, является ли неизвестный объект элементарным значением или объектом. Если это объект, то посредством свойства constructor можно определить, каков тип этого объекта. Например, следующая функция определяет, является ли данное значение массивом:

```
function isArray(x) {
    return ((typeof x == "object") && (x.constructor == Array));
}
```

Заметьте, однако, что хотя этот прием эффективен для объектов, встроенных в базовый JavaScript, его работа с «хост-объектами», такими как объект Window клиентского JavaScript, не гарантирована. Реализация по умолчанию метода Object.toString() предоставляет еще один способ определения типа неизвестного объекта.

Смотри также

Object.toString()

Object.hasOwnProperty()

JavaScript 1.5; JScript 5.5; ECMAScript v3

проверяет, является ли свойство унаследованным

Синтаксис

объект.hasOwnProperty(*имя_свойства*)

Аргументы

имя_свойства

Строка, содержащая имя свойства *объекта*.

Возвращаемое значение

true, если *объект* имеет неунаследованное свойство с именем, заданным в *имени_свойства*. Возвращает false, если *объект* не имеет свойства с указанным именем или если он наследует это свойство от своего объекта-прототипа.

Описание

В главе 8 говорится, что объекты JavaScript могут иметь собственные свойства, а также могут наследовать свойства от их объекта-прототипа. Метод hasOwnProperty() предоставляет способ, позволяющий установить различия между унаследованными свойствами и неунаследованными локальными свойствами.

Пример

```
var o = new Object();          // Создаем объект
o.x = 3.14;                    // Определяем неунаследованное локальное
                               // свойство
o.hasOwnProperty("x");         // Возвращает true: x - это локальное
                               // свойство o
o.hasOwnProperty("y");         // Возвращает false: o не имеет свойства y
o.hasOwnProperty("toString");  // Возвращает false: свойство toString
                               // унаследовано
```

Смотри также

Function.prototype, Object.propertyIsEnumerable(); главу 8

Object.isPrototypeOf()

JavaScript 1.5; JScript 5.5; ECMAScript v3

является ли один объект прототипом другого объекта

Синтаксис

объект.isPrototypeOf(*o*)

Аргументы

o – любой объект.

Возвращаемое значение

true, если *объект* представляет собой прототип *o*. Возвращает false, если *o* не является объектом или если *объект* не является прототипом *o*.

Описание

Как объяснялось в главе 8, объекты JavaScript наследуют свойства от их объекта-прототипа. К прототипу объекта можно обращаться с помощью свойства prototype функции-конструктора, которая применялась для создания и инициализации объекта. Метод isPrototypeOf() предоставляет способ, позволяющий определить, является ли один объект прототипом другого. Этот прием может применяться для определения класса объекта.

Пример

```
var o = new Object();                          // Создание объекта
Object.prototype.isPrototypeOf(o)              // true: o - это объект
Function.prototype.isPrototypeOf(o.toString);  // true: toString - это функция
Array.prototype.isPrototypeOf([1,2,3]);       // true: [1,2,3] - это массив

// Ту же проверку можно выполнить другим способом
(o.constructor == Object);  // true: o был создан с помощью конструктора
                            // Object()
(o.toString.constructor == Function);  // true: o.toString - это функция

// Объекты-прототипы сами имеют прототипы. Следующий вызов возвращает
// true, показывая, что объекты-функции наследуют свойства
// от Function.prototype, а также от Object.prototype.
Object.prototype.isPrototypeOf(Function.prototype);
```

Смотри также

Function.prototype, Object.constructor; главу 8

Object.propertyISEnumerable()

JavaScript 1.5; JScript 5.5; ECMAScript v3

будет ли свойство видимо для цикла for/in?

Синтаксис

объект.propertyIsEnumerable(*имя_свойства*)

Аргументы

имя_свойства

Строка, содержащая имя свойства *объекта*.

Возвращаемое значение

`true`, если у *объекта* есть неунаследованное свойство с именем, указанным в аргументе *имя_свойства*, и если это свойство «перечисляемое», то есть будет перечислено циклом `for/in` для *объекта*.

Описание

Инструкция `for/in` выполняет цикл по «перечисляемым» свойствам объекта. Однако не все свойства объекта являются перечисляемыми: свойства, добавленные в объект кодом JavaScript, перечисляемы, а предопределенные свойства (например, методы) встроенных объектов обычно не перечисляемы. Метод `propertyIsEnumerable()` предоставляет способ для установления различия между перечисляемыми и неперечисляемыми свойствами. Однако заметьте, спецификация ECMAScript утверждает, что `propertyIsEnumerable()` не проверяет цепочку прототипов, то есть этот метод годится только для локальных свойств объекта и не предоставляет способа для проверки перечисляемости унаследованных свойств.

Пример

```
var o = new Object();                        // Создает объект
o.x = 3.14;                                  // Определяет свойство
o.propertyIsEnumerable("x");                 // true: свойство x локальное и перечисляемое
o.propertyIsEnumerable("y");                 // false: o не имеет свойства y
o.propertyIsEnumerable("toString");          // false: свойство toString является унаследованным
Object.prototype.propertyIsEnumerable("toString");    // false: неперечисляемое
```

Ошибки

Спецификация ограничивает `propertyIsEnumerable()` только проверкой неунаследованных свойств, и это, несомненно, ошибка. Internet Explorer 5.5 реализует этот метод в соответствии со спецификацией. Netscape 6.0 реализует его так, что рассматривается цепочка прототипов. Возможно, что схема работы этого метода предполагалась именно такой, но это нарушает спецификацию, и Netscape 6.1 был изменен, чтобы соответствовать IE 5.5. По причине ошибки в спецификации этот метод менее полезен, чем мог бы быть.

Смотри также

Function.prototype, Object.hasOwnProperty(); главу 8

Object.toLocaleString()

JavaScript 1.5; JScript 5.5; ECMAScript v3

возвращает локализованное строковое представление объекта

Синтаксис

объект.`toString()`

Возвращаемое значение

Строковое представление объекта.

Описание

Этот метод предназначен для получения строкового представления объекта, локализованного в соответствии с текущими региональными настройками. Метод `toLocaleString()`, предоставляемый по умолчанию классом Object, просто вызывает метод

toString() и возвращает полученную от него нелокализованную строку. Однако обратите внимание, что другие классы, в том числе Array, Date и Number, определяют собственные версии этого метода для выполнения локализованного преобразования в строку. Определяя собственные классы, вы можете также перекрыть этот метод.

Смотри также

Array.toLocaleString(), Date.toLocaleString(), Number.toLocaleString(), Object.toString()

Object.toString() JavaScript 1.0; JScript 2.0; ECMAScript v1

определяет строковое представление объекта

Синтаксис

объект.toString()

Возвращаемое значение

Строка, представляющая объект.

Описание

Метод toString() не относится к тем, которые часто вызываются в программах на JavaScript явно. Программист определяет этот метод в своих объектах, а система вызывает метод, когда требуется преобразовать объект в строку.

Система JavaScript вызывает метод toString() для преобразования объекта в строку всякий раз, когда объект используется в строковом контексте. Например, если объект преобразуется в строку при передаче в функцию, требующую строкового аргумента:

```
alert(my_object);
```

Подобным же образом объекты преобразуются в строки, когда они конкатенируются со строками с помощью оператора +:

```
var msg = 'Мой объект: ' + my_object;
```

Метод toString() вызывается без аргументов и должен возвращать строку. Для того чтобы от строки, возвращаемой из этого метода, была какая-то польза, эта строка должна каким-либо образом базироваться на значении объекта, для которого был вызван метод.

Определяя в JavaScript специальный класс, целесообразно определить для него метод toString(). Если этого не сделать, объект наследует метод toString(), определенный по умолчанию в классе Object. Этот стандартный метод возвращает строку в формате:

```
[object класс]
```

где *класс* – это класс объекта: значение вроде «Object», «String», «Number», «Function», «Window», «Document» и т. д. Это поведение стандартного метода toString() иногда бывает полезно для определения типа или класса неизвестного объекта. Однако большинство объектов имеют собственную версию toString(), поэтому для произвольного объекта o необходимо явно вызывать метод Object.toString() с помощью следующего кода:

```
Object.prototype.toString.apply(o);
```

Обратите внимание, что этот способ идентификации неизвестных объектов годится только для встроенных объектов. Если вы определяете свой собственный класс объектов, то *класс* для него будет соответствовать значению «Object». В этом случае дополнительную информацию об объекте позволит получить свойство `Object.constructor`.

Метод `toString()` может быть очень полезен при отладке программ JavaScript – он позволяет печатать объекты и видеть их значения. По одной только этой причине есть смысл определять метод `toString()` для каждого создаваемого вами класса.

Несмотря на то что метод `toString()` обычно вызывается системой автоматически, бывают случаи, когда его можно вызывать самостоятельно. Например, когда требуется выполнить явное преобразование объекта в строку, если JavaScript не делает это автоматически:

```
y = Math.sqrt(x);      // Вычисляет число
ystr = y.toString();  // Преобразует его в строку
```

В этом примере обратите внимание на то, что числа имеют встроенный метод `toString()`, обеспечивающий принудительное преобразование.

В других обстоятельствах предпочтительным может оказаться метод `toString()` – даже в таком контексте, когда JavaScript выполняет преобразование автоматически. Применение `toString()` явно может сделать код более понятным:

```
alert(my_obj.toString());
```

Смотри также

Object.constructor(), Object.toLocaleString(), Object.valueOf()

Object.valueOf()

JavaScript 1.1; JScript 2.0; ECMAScript v1

элементарное значение указанного объекта

Синтаксис

объект`.valueOf()`

Возвращаемое значение

Элементарное значение, связанное с *объектом*, если оно есть. Если с *объектом* не связано значение, метод возвращает сам объект.

Описание

Метод `valueOf()` объекта возвращает элементарное значение, связанное с этим объектом, если оно есть. Для объектов типа Object элементарное значение отсутствует, и его метод возвращает сам объект.

Однако для объектов типа Number `valueOf()` возвращает элементарное числовое значение, представляемое объектом. Аналогично, он возвращает элементарное логическое значение, связанное с объектом Boolean, и строку, связанную с объектом String.

Программисту редко приходится самостоятельно вызывать метод `valueOf()`. JavaScript делает это автоматически всякий раз, втретив объект там, где ожидается элементарное значение. Из-за автоматического вызова метода `valueOf()` фактически трудно даже провести различие между элементарными значениями и соответствующими им объектами. Оператор `typeof`, например, показывает различие между строками и объектами String, но с практической точки зрения они работают в коде JavaScript эквивалентным образом.

Метод valueOf() объектов Number, Boolean и String преобразует эти объекты-обертки в представляемые ими элементарные значения. Конструктор Object() выполняет противоположную операцию при вызове с числовым, логическим или строковым аргументом: он «заворачивает» элементарное значение в соответствующий объект-обертку. В большинстве случаев JavaScript берет это преобразование «элементарное значение-объект» на себя, поэтому необходимость в таком вызове конструктора Object() возникает редко.

Иногда программисту требуется определить специальный метод valueOf() для собственных объектов. Например, определить объектный тип JavaScript для представления комплексных чисел (вещественное число плюс мнимое число). Как часть этого объектного типа можно определить методы для выполнения комплексного сложения, умножения и т. д. Но еще может потребоваться возможность рассматривать комплексные числа как обычные вещественные путем отбрасывания мнимой части. Для этого можно сделать примерно следующее:

```
Complex.prototype.valueOf = new Function("return this.real");
```

Определив метод valueOf() для собственного объектного типа Complex, можно, например, передавать объекты комплексных чисел в функцию Math.sqrt(), которая вычислит квадратный корень из вещественной части комплексного числа.

Смотри также

Object.toString()

parseFloat() JavaScript 1.0; JScript 1.0; ECMAScript v1

преобразование строки в число

Синтаксис

parseFloat(*s*)

Аргументы

s – строка, которая должна быть разобрана и преобразована в число.

Возвращаемое значение

Возвращает выделенное из строки число или NaN, если *s* не начинается с допустимого числа. В JavaScript 1.0, когда *s* не может быть преобразовано в число, parseFloat() возвращает 0 вместо NaN.

Описание

Функция parseFloat() разбирает строку и возвращает первое число, найденное в *s*. Разбор прекращается и значение возвращается, когда parseFloat() встречает в *s* символ, который не является допустимой частью числа. Если *s* не начинается с числа, которое parseFloat() может разобрать, функция возвращает значение «нечисло» NaN. Проверка на это возвращаемое значение выполняется функцией isNaN(). Для того чтобы выделить только целую часть числа, применяется parseInt(), а не parseFloat().

Ошибки

NaN в JavaScript 1.0 не поддерживается, поэтому в данной версии языка parseFloat() возвращает 0, когда не может разобрать *s*. Это значит, что если возвращаемое значение parseFloat() равно 0, то в JavaScript 1.0 необходимы дополнительные проверки строки *s*, позволяющие определить, действительно ли строка представляет 0 или вообще не представляет число.

Смотри также

isNaN(), parseInt()

parseInt()

JavaScript 1.0; JScript 1.0; ECMAScript v1

преобразует строку в целое

Синтаксис

parseInt(*s*)

parseInt(*s*, *основание*)

Аргументы

s – разбираемая строка.

основание

Необязательный строковый аргумент, представляющий основание системы счисления анализируемого числа. Если этот аргумент отсутствует или равен 0, извлекается десятичное или шестнадцатеричное (если число начинается с «0x» или «0X») число. Если этот аргумент меньше 2 или больше 36, parseInt() возвращает NaN.

Возвращаемое значение

Извлекаемое число (NaN, если *s* не начинается с корректного целого). В JavaScript 1.0, если строка *s* не может быть разобрана, parseInt() возвращает 0 вместо NaN.

Описание

Функция parseInt() разбирает и возвращает первое число (с необязательным ведущим знаком «минус»), найденное в *s*. Разбор останавливается и значение возвращается, когда parseInt() встречает в *s* символ, не являющийся допустимой цифрой для указанного *основания*. Если *s* не начинается с числа, которое может быть разобрано parseInt(), функция возвращает значение «нечисло» NaN. Проверка на это возвращаемое значение выполняется функцией isNaN().

Аргумент *основание* задает основание извлекаемого числа. При *основании*, равном 10, parseInt() извлекает десятичное число. Если этот аргумент равен 8, то извлекается восьмеричное число (состоящее из цифр от 0 до 7), а если 16, то шестнадцатеричное (цифры от 0 до 9 и буквы от A до F). Аргумент *основание* может быть любым числом от 2 до 36.

Если *основание* равно 0 или не указано, parseInt() пытается определить систему счисления по строке *s*. Если *s* начинается (после необязательного знака «минус») с 0x, parseInt() разбирает оставшуюся часть *s* как шестнадцатеричное число. Если *s* начинается с 0, стандарт ECMAScript v3 разрешает интерпретировать следующие символы либо как восьмеричное, либо как десятичное число. В противном случае, если *s* начинается с цифры от 1 до 9, parseInt() разбирает строку как десятичное число.

Пример

```
parseInt("19", 10);  // Возвращает 19  (10 + 9)
parseInt("11", 2);   // Возвращает 3   (2 + 1)
parseInt("17", 8);   // Возвращает 15  (8 + 7)
parseInt("1f", 16);  // Возвращает 31  (16 + 15)
parseInt("10");      // Возвращает 10
parseInt("0x10");    // Возвращает 16
parseInt("010");     // Неоднозначно: возвращает либо 10, либо 8
```

Ошибки

Когда *основание* не указано, ECMAScript v3 разрешает разбирать строку, начинающуюся с «0» (но не с «0x» или «0X»), либо как восьмеричное, либо как десятичное число. Во избежание двусмысленности необходимо явно указать основание или не указывать его только в тех случаях, когда наверняка известно, что все разбираемые числа будут десятичными или шестнадцатеричными, начинающимися с префикса «0x» или «0X».

В JavaScript 1.0 `NaN` не поддерживается, а `parseInt()` возвращает 0 вместо `NaN`, когда не может выполнить разбор *s*. В этой версии языка `parseInt()` не может провести различие между некорректными входными данными и корректным аргументом «0».

Смотри также

isNaN(), parseFloat()

RangeError

JavaScript 1.5; JScript 5.5; ECMAScript v3

генерируется, когда число выходит из допустимого диапазона **наследуется от Error**

Конструктор

`new RangeError()`

`new RangeError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он используется в качестве значения свойства `message` объекта RangeError.

Возвращаемое значение

Только что созданный объект RangeError. Если аргумент *сообщение* указан, то для объекта RangeError он будет выступать в качестве значения свойства `message`; в противном случае RangeError возьмет в качестве значения этого свойства строку по умолчанию, определенную в реализации. Конструктор `RangeError()`, вызываемый как функция, без оператора `new`, ведет себя точно так же, как если бы был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробнее см. в статье «Error.message».

`name`

Строка, задающая тип исключения. Все объекты RangeError наследуют для этого свойства значение «RangeError».

Описание

Экземпляр класса RangeError создается, когда числовое значение оказывается вне допустимого диапазона. Например, установка длины массива равной отрицательному числу приводит к генерации RangeError. Подробности о генерации и перехвате исключений см. в статье «Error».

Смотри также

Error, Error.message, Error.name

ReferenceError

JavaScript 1.5; JScript 5.5; ECMAScript v3

генерируется, когда читается несуществующая переменная **наследуется от Error**

Конструктор

`new ReferenceError()`

`new ReferenceError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта ReferenceError.

Возвращаемое значение

Только что созданный объект ReferenceError. Если аргумент *сообщение* указан, объект ReferenceError возьмет его в качестве значения своего свойства `message`; в противном случае он возьмет строку по умолчанию, определенную в реализации. Конструктор `ReferenceError()`, вызывамый как функция, без оператора `new`, ведет себя точно так же, как если бы был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробнее см. в статье «Error.message».

`name`

Строка, задающая тип исключения. Все объекты ReferenceError наследуют для этого свойства значение «ReferenceError».

Описание

Экземпляр класса ReferenceError создается при попытке прочитать значение несуществующей переменной. О генерации и перехвате исключений см. статью «Error».

Смотри также

Error, Error.message, Error.name

RegExp

JavaScript 1.2; JScript 3.0; ECMAScript v3

регулярные выражения для поиска по маске

Синтаксис литерала

/маска/атрибуты

Конструктор

`new RegExp(`*маска*`, `*атрибуты*`)`

Аргументы

маска

Строка, задает маску регулярного выражения или другое регулярное выражение.

атрибуты

Необязательная строка, содержащая любые из атрибутов «g», «i» и «m», задающие глобальный, нечувствительный к регистру и многострочный поиск. До стандартизации ECMAScript атрибут «m» не был доступен. Если аргумент *маска* – это регулярное выражение, а не строка, данный аргумент может отсутствовать.

Возвращаемое значение

Новый объект `RegExp` с указанной маской и флагами. Если аргумент *маска* представляет собой регулярное выражение, а не строку, конструктор `RegExp()` создает новый объект RegExp, используя ту же маску и флаги, что и в указанном RegExp. Если `RegExp()` вызывается как функция без оператора `new`, то ведет себя точно так же, как с оператором `new`, кроме случая, когда *маска* уже является объектом RegExp; тогда функция возвращает аргумент *маска*, вместо того чтобы создавать новый объект RegExp.

Исключения

`SyntaxError`

Если *маска* не является допустимым регулярным выражением или если аргумент *атрибуты* содержит символы, отличные от «g», «i» и «m».

`TypeError`

Если *маска* – это объект RegExp, и присутствует аргумент *атрибуты*.

Свойства экземпляра

`global`

Имеется ли в RegExp атрибут `g`.

`ignoreCase`

Имеется ли в RegExp атрибут `i`.

`lastIndex`

Позиция символа при последнем нахождении соответствия; используется для нахождения в строке нескольких соответствий.

`multiline`

Имеется ли в RegExp атрибут `m`.

`source`

Исходный текст регулярного выражения.

Методы

`exec()`

Выполняет мощный универсальный поиск по маске.

`test()`

Проверяет, содержит ли строка данную маску.

Описание

Объект RegExp представляет регулярное выражение – мощное средство для выполнения поиска по маске в строках. Синтаксис и применение регулярных выражений полностью описаны в главе 10.

Смотри также

Главу 10

RegExp.exec() JavaScript 1.2; JScript 3.0; ECMAScript v3

универсальный поиск по маске

Синтаксис

regexp.exec(*строка*)

Аргументы

строка

Строка, в которой выполняется поиск.

Возвращаемое значение

Массив, содержащий результаты поиска или `null`, если соответствие не найдено. Формат возвращаемого массива описан ниже.

Исключения

`TypeError`

Если метод вызывается для объекта, не являющегося RegExp.

Описание

`exec()` – это наиболее мощный из всех методов объектов RegExp и String для поиска по маске. Это универсальный метод, применять который несколько сложнее, чем `RegExp.test()`, `String.search()`, `String.replace()` и `String.match()`.

`exec()` ищет в *строке* текст, соответствующий выражению *regexp*. И если находит, то возвращает массив результатов; в противном случае он возвращает `null`. Элемент 0 полученного массива представляет собою искомый текст. Элемент 1 – это текст, соответствующий первому подвыражению в скобках внутри *regexp*, если оно есть. Элемент 2 соответствует второму подвыражению и т. д. Свойство массива `length` указывает, как обычно, количество элементов в массиве. В дополнение к элементам массива и свойству `length` в значении, возвращаемое `exec()`, есть еще два свойства. Свойство `index` указывает позицию первого символа искомого текста. Свойство `input` ссылается на *строку*. Этот возвращаемый массив совпадает с массивом, возвращаемым методом `String.match()`, когда он вызывается для неглобального объекта RegExp.

Когда exec() вызывается для неглобальной маски, он выполняет поиск и возвращает результат, описанный выше. Однако если *regexp* – глобальное регулярное выражение, exec() ведет себя несколько более сложным образом. Он начинает поиск в *строке* с символьной позиции, заданной свойством *regexp*.lastIndex. Найдя соответствие, метод устанавливает lastIndex равным позиции первого символа после найденного соответствия. Это значит, что exec() можно вызвать несколько раз, чтобы выполнить цикл по всем соответствиям в строке. Если метод exec() больше не может найти соответствий, он возвращает null и сбрасывает lastIndex в ноль. Начиная поиск непосредственно после успешного нахождения соответствия в другой строке, необходимо соблюдать внимательность и вручную установить lastIndex равным нулю.

Обратите внимание, что exec() всегда включает все детали для найденного соответствия в возвращаемый им массив независимо от того, является ли *regexp* глобальной маской. Этим exec() отличается от String.match(), который возвращает намного меньше информации при работе с глобальными масками. Вызов exec() в цикле – единственный способ получить полную информацию о результатах поиска для глобальной маски.

Пример

Можно запускать exec() в цикле для нахождения всех соответствий в строке:

```
var pattern = /\bJava\w*\b/g;
var text = "JavaScript is more fun than Java or JavaBeans!";
var result;
while((result = pattern.exec(text)) != null) {
    alert("Matched `" + result[0] +
          "' at position " + result.index +
          " next search begins at position " + pattern.lastIndex);
}
```

Ошибки

В JScript 3.0 exec() не устанавливает и не использует свойство lastIndex корректным образом, поэтому этот метод не может применяться с глобальными масками в циклах, как показано в приведенном выше примере.

Смотри также

RegExp.lastIndex, RegExp.test(), String.match(), String.replace(), String.search(); главу 10

RegExp.global

JavaScript 1.2; JScript 5.5; ECMAScript v3

выполняется ли глобальный поиск по данному регулярному выражению

Синтаксис

regexp.global

Описание

global – это логическое свойство объектов RegExp, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение глобальный поиск, то есть было ли оно создано с атрибутом g.

RegExp.ignoreCase

JavaScript 1.2; JScript 5.5; ECMAScript v3

чувствительно ли регулярное выражение к регистру

Синтаксис

regexp.ignoreCase

Описание

ignoreCase — это логическое свойство объектов RegExp, доступное только для чтения. Оно указывает, выполняет ли данное регулярное выражение поиск, нечувствительный к регистру, то есть было ли оно создано с атрибутом i.

RegExp.lastIndex

JavaScript 1.2; JScript 5.5; ECMAScript v3

начальная позиция следующего поиска

Синтаксис

regexp.lastIndex

Описание

lastIndex — это доступное для чтения и записи свойство объектов RegExp. Для регулярных выражений с установленным атрибутом g оно содержит целое, указывающее позицию в строке, которая непосредственно следует за последним соответствием, найденным с помощью методов RegExp.exec() и RegExp.test(). Эти методы используют данное свойство в качестве начальной точки следующего выполняемого ими поиска. Благодаря этому данные методы можно вызывать повторно, для выполнения цикла по всем соответствиям в строке. Обратите внимание, что lastIndex не используется объектами RegExp, не имеющими атрибута g и не представляющими собой глобальные маски.

Это свойство доступно для чтения и для записи, поэтому можно установить его в любой момент, чтобы указать, где в целевой строке должен быть начат следующий поиск. exec() и test() автоматически сбрасывают lastIndex в 0, когда не могут найти какого-либо (или следующего) соответствия. Начиная поиск в новой строке после успешного поиска в предыдущей, необходимо явно установить это свойство равным 0.

Смотри также

RegExp.exec(), RegExp.test()

RegExp.source

JavaScript 1.2; JScript 3.0; ECMAScript v3

текст регулярного выражения

Синтаксис

regexp.source

Описание

source — доступное только для чтения строковое свойство объектов RegExp, содержащее текст маски RegExp. Текст не включает ограничивающие наклонные черты, используемые в литералах регулярных выражений, и не включает атрибуты g, i и m.

RegExp.test()

JavaScript 1.2; JScript 3.0; ECMAScript v3

проверяет, соответствует ли строка маске

Синтаксис

regexp.test(*строка*)

Аргументы

строка

Проверяемая строка.

Возвращаемое значение

true, если *строка* содержит текст, соответствующий *regexp*; false в противном случае.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося RegExp.

Описание

test() проверяет *строку*, чтобы увидеть, содержит ли она текст, который соответствует *regexp*. Если да, он возвращает true; в противном случае он возвращает false. Вызов метода test для регулярного выражения *r* и передача ему строки s эквивалентны следующему выражению:

```
(r.exec(s) != null)
```

Пример

```
var pattern = /java/i;
pattern.test("JavaScript");   // Возвращает true
pattern.test("ECMAScript");   // Возвращает false
```

Смотри также

RegExp.exec(), RegExp.lastIndex, String.match(), String.replace(), String.substring(); главу 10

RegExp.toString()

JavaScript 1.2; JScript 3.0; ECMAScript v3

преобразует регулярное выражение в строку **перекрывает Object.toString()**

Синтаксис

regexp.toString()

Возвращаемое значение

Строковое представление *regexp*.

Исключения

TypeError

Если метод вызывается для объекта, который не является RegExp.

Описание

Метод RegExp.toString() возвращает строковое представление регулярного выражения в форме литерала регулярного выражения.

Обратите внимание, что от реализаций не требуется обязательного добавления управляющих последовательностей, гарантирующих, что возвращаемая строка является корректным литералом регулярных выражений. Рассмотрим регулярное выражение, созданное с помощью следующего выражения: `new RegExp("/", "g")`. Реализация `RegExp.toString()` может вернуть для регулярного выражения `///g` либо добавить управляющую последовательность и вернуть `/\//g`.

String

JavaScript 1.0; JScript 1.0; ECMAScript v1

поддержка строк **наследуется от Object**

Конструктор

```
new String(s)  // функция-конструктор
String(s)      // функция преобразования
```

Аргументы

s Значение, подлежащее сохранению в объекте String или преобразованию в элементарное строковое значение.

Возвращаемое значение

Когда `String()` вызывается в качестве конструктора с оператором `new`, функция возвращает объект String, содержащий строку *s* или строковое представление *s*. Конструктор `String()`, вызванный без оператора `new`, преобразует *s* в элементарное строковое значение и возвращает преобразованное значение.

Свойства

`length`

Количество символов в строке.

Методы

`charAt()`

Извлекает из строки символ, находящийся в указанной позиции.

`charCodeAt()`

Возвращает код символа, находящегося в указанной позиции.

`concat()`

Выполняет конкатенацию одного или нескольких значений со строкой.

`indexOf()`

Выполняет поиск символа или подстроки в строке.

`lastIndexOf()`

Выполняет поиск символа или подстроки в строке с конца.

`match()`

Выполняет поиск по маске с помощью регулярного выражения.

`replace()`

Выполняет операцию поиска-замены с помощью регулярного выражения.

`search()`

Ищет в строке подстроку, соответствующую регулярному выражению.

`slice()`

Возвращает фрагмент или подстроку в строке.

`split()`

Разбивает строку на массив строк по указанной строке-разделителю или регулярному выражению.

`substring()`

Извлекает подстроку из строки.

`substr()`

Извлекает подстроку из строки. Вариант `substring()`.

`toLowerCase()`

Возвращает копию строки, в которой все символы переведены в нижний регистр.

`toString()`

Возвращает элементарное строковое значение.

`toUpperCase()`

Возвращает копию строки, в которой все символы переведены в верхний регистр.

`valueOf()`

Возвращает элементарное строковое значение.

Статические методы

`String.fromCharCode()`

Создает новую строку, помещая в нее принятые в качестве аргументов коды символов.

HTML-методы

Начиная с JavaScript 1.0 и JScript 1.0 в классе String определено несколько методов, которые возвращают строку, измененную путем добавления к ней тегов HTML. Эти методы никогда не были стандартизованы в ECMAScript, но позволяют динамически генерировать код HTML и в клиентском и в серверном коде JavaScript. Если вы готовы к применению нестандартных методов, то можете создать исходный текст HTML для гиперссылки, выделенной полужирным шрифтом красного цвета, с помощью следующего кода:

```
var s = "нажми сюда!";
var html = s.bold().link("javascript:alert('hello')").fontcolor("red");
```

Эти методы не стандартизованы, поэтому для них отсутствуют индивидуальные справочные статьи:

`anchor(`*имя*`)`

Возвращает копию строки в окружении тега `<a name=>`.

`big()`

Возвращает копию строки в окружении тега `<big>`.

`blink()`

Возвращает копию строки в окружении тега `<blink>`.

`bold()`

Возвращает копию строки в окружении тега `<b>`.

`fixed()`

Возвращает копию строки в окружении тега `<tt>`.

`fontcolor(цвет)`

Возвращает копию строки в окружении тега `<font color=>`.

`fontsize(размер)`

Возвращает копию строки в окружении тега `<font size=>`.

`italics()`

Возвращает копию строки в окружении тега `<i>`.

`link(url)`

Возвращает копию строки в окружении тега `<a href=>`.

`small()`

Возвращает копию строки в окружении тега `<small>`.

`strike()`

Возвращает копию строки в окружении тега `<strike>`.

`sub()`

Возвращает копию строки в окружении тега `<sub>`.

`sup()`

Возвращает копию строки в окружении тега `<sup>`.

Описание

Строки – это элементарный тип данных в JavaScript. Класс String предназначен для предоставления методов для работы с элементарными строковыми значениями. Свойство `length` объекта String указывает количество символов в строке. Класс String определяет немало методов для работы со строками. Например, имеются методы для извлечения символа или подстроки или для поиска символа или подстроки. Обратите внимание, что строки JavaScript *неизменяемы* – ни один из методов, определенных в классе String, не позволяет изменять содержимое строки. Зато методы, подобные `String.toUpperCase()`, возвращают абсолютно новую строку, не изменяя исходную.

В реализациях Netscape JavaScript 1.2 и более поздних строки ведут себя как массивы символов, доступные только для чтения. Например, чтобы извлечь третий символ из строки `s`, можно написать `s[2]` вместо более стандартного `s.charAt(2)`. Кроме того, инструкция `for/in`, примененная к строке, перечисляет эти индексы массива для каждого символа в строке. (Заметьте, однако, что свойство `length` не перечисляется, как должно быть по спецификации ECMAScript.) Это поведение нестандартное, и его следует избегать.

Смотри также

Главу 3

String.charAt()

JavaScript 1.0; JScript 1.0, ECMAScript v1

получает *n*-й символ из строки

Синтаксис

строка.charAt(*n*)

Аргументы

n – индекс символа, который должен быть извлечен из *строки*.

Возвращаемое значение

n-й символ из *строки*.

Описание

String.charAt() возвращает *n*-й символ из *строки*. Номер первого символа в строке равен нулю. Если *n* не находится между 0 и *string.length*–1, этот метод возвращает пустую строку. Заметьте, что в JavaScript нет символьного типа данных, отличного от строкового, поэтому извлеченный символ представляет собой строку длиной 1.

Смотри также

String.charCodeAt(), String.indexOf(), String.lastIndexOf()

String.charCodeAt()

JavaScript 1.2; JScript 5.5; ECMAScript v1

получает код *n*-го символа из строки

Синтаксис

строка.charCodeAt(*n*)

Аргументы

n – индекс символа, код которого должен быть получен.

Возвращаемое значение

Код Unicode *n*-го символа в *строке*. Возвращает 16-разрядное целое между 0 и 65 535.

Описание

charCodeAt() – это то же самое, что charAt(), за исключением того, что этот метод возвращает код символа, находящегося в определенной позиции, а не подстроку, содержащую сам символ. Если *n* отрицательно или меньше или равно длине строки, charCodeAt() возвращает NaN.

Создание строки по коду Unicode символа описано в статье «String.fromCharCode()».

Ошибки

В JavaScript 1.2 (например, реализованном в Netscape 4.0) нет полной поддержки 16-битовых символов и строк Unicode.

Смотри также

String.charAt(), String.fromCharCode()

String.concat() JavaScript 1.2; JScript 3.0; ECMAScript v3

объединяет строки

Синтаксис

строка.concat(*значение*, ...)

Аргументы

значение, ...

Одно или более значений, объединяемых со строкой *string*.

Возвращаемое значение

Новая строка, полученная при объединении всех аргументов со *строкой*.

Описание

concat() преобразует все свои аргументы в строки (если это нужно) и добавляет их по порядку в конец *строки*. Возвращает полученную объединенную строку. Обратите внимание, что сама *строка* не меняется.

String.concat() представляет собой аналог Array.concat(). Заметьте, что конкатенацию строк часто проще выполнить с помощью оператора +.

Смотри также

Array.concat()

String.fromCharCode() JavaScript 1.2; JScript 3.0; ECMAScript v1

создает строку из кодов символов

Синтаксис

String.fromCharCode(*c1*, *c2*, ...)

Аргументы

c1, *c2*, ...

Ноль и более целых значений, задающих коды Unicode для символов создаваемой строки.

Возвращаемое значение

Новая строка, содержащая символы с указанными кодами.

Описание

Этот статический метод предоставляет способ создания строки путем задания индивидуальных числовых кодов ее символов. Заметьте, что статический метод fromCharCode() представляет собой свойство конструктора String() и фактически не является методом строк или объектов String.

String.charCodeAt() — это метод экземпляра, являющийся компаньоном этого метода и предоставляющий способ получения кодов индивидуальных символов строки.

Пример

```
// Создает строку "hello"
var s = String.fromCharCode(104, 101, 108, 108, 111);
```

Ошибки

В JavaScript 1.2 (например, в Netscape 4.0) нет полной поддержки 16-битовых символов и строк Unicode.

Смотри также

String.charCodeAt()

String.indexOf()

JavaScript 1.0; JScript 1.0, ECMAScript v1

ищет строку

Синтаксис

строка.indexOf(*подстрока*)

строка.indexOf(*подстрока*, *start*)

Аргументы

подстрока

Подстрока, которая должна быть найдена в *строке*.

start

Необязательный целый аргумент, задающий позицию в *строке*, с которой следует начать поиск. Допустимые значения: от 0 (позиция первого символа в строке) до *string*.length –1 (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с первого символа строки.

Возвращаемое значение

Позиция первого вхождения *подстроки* внутри *строки*, присутствующая после позиции *start*, если *подстрока* найдена, или –1, если такое вхождение не найдено.

Описание

String.indexOf() выполняет поиск в *строке* от начала к концу, чтобы увидеть, содержит ли она *подстроку*. Поиск начинается с позиции *start* в *строке* или с начала *строки*, если аргумент *start* не указан. Если *подстрока* найдена, String.indexOf() возвращает позицию первого символа первого вхождения *подстроки* в *строке*. Позиции символов в *строке* нумеруются с нуля.

Если *подстрока* в *строке* не найдена, String.indexOf() возвращает –1.

Ошибки

В JavaScript 1.0 и 1.1, если *start* больше длины *строки*, indexOf() возвращает пустую строку, а не –1.

Смотри также

String.charAt(), String.lastIndexOf(), String.substring()

String.lastIndexOf()

JavaScript 1.0; JScript 1.0, ECMAScript v1

выполняет поиск в строке с конца

Синтаксис

строка.lastIndexOf(*подстрока*)

строка.lastIndexOf(*подстрока*, *start*)

Аргументы

подстрока

Подстрока, которая должна быть найдена в *строке*.

start

Необязательный целый аргумент, задающий позицию в *строке*, с которой следует начать поиск. Допустимые значения: от 0 (позиция первого символа в строке) до *string*.length–1 (позиция последнего символа в строке). Если этот аргумент отсутствует, поиск начинается с последнего символа строки.

Возвращаемое значение

Позиция последнего вхождения *подстроки* в *строке*, начиная с позиции *start*, если подстрока найдена, или –1, если такое вхождение не найдено.

Описание

String.lastIndexOf() просматривает строку от конца к началу, чтобы увидеть, содержит ли она *подстроку*. Поиск выполняется с позиции *start* внутри *строки* или с конца строки, если аргумент *start* не указан. Если *подстрока* найдена, String.lastIndexOf() возвращает позицию первого символа этого вхождения. Метод выполняет поиск от конца к началу, поэтому первое найденное вхождение является последним в строке, расположенным до позиции *start*.

Если *подстрока* не найдена, String.lastIndexOf() возвращает –1.

Заметьте, что хотя String.lastIndexOf() ищет *строку* от конца к началу, он все равно нумерует позиции символов в *строке* с начала. Первый символ строки занимает позицию с номером 0, а последний – *string*.length–1.

Смотри также

String.charAt(), String.indexOf(), String.substring()

String.length

JavaScript 1.0; JScript 1.0, ECMAScript v1

длина строки

Синтаксис

строка.length

Описание

Свойство String.length – это доступное только для чтения целое, указывающее количество символов с указанной *строке*. Для любой строки *s* индекс последнего символа равен *s*.length–1. Свойство length строки не перечисляется циклом for/in и не может быть удалено с помощью оператора delete.

String.localeCompare()

JavaScript 1.5; JScript 5.5; ECMAScript v3

сравнивает одну строку с другой, используя региональную сортировку

Синтаксис

строка.localeCompare(*целевая_строка*)

Аргументы

целевая_строка

Строка, сравниваемая со строкой с учетом региона.

Возвращаемое значение

Число, обозначающее результат сравнения. Если *строка* «меньше» *целевой_строки*, localeCompare() возвращает отрицательное число. Если *строка* «больше» *целевой_строки*, метод возвращает положительное число. Если строки идентичны или неотличимы в соответствии с региональными соглашениями о сортировке, метод возвращает 0.

Описание

Когда к строкам применяются операторы < и >, сравнение выполняется только по кодам этих символов в Unicode, порядок сортировки, принятый в текущем регионе, не учитывается. Сортировка, выполняемая подобным образом, не всегда оказывается верной. Возьмем, например, испанский язык, в котором буквы «ch» сортируются как одна буква, расположенная между буквами «c» и «d».

localeCompare() предоставляет способ сравнения строк, учитывающий порядок сортировки из региональных настроек по умолчанию. Стандарт ECMAScript не определяет, как должно выполняться сравнение с учетом региона; в нем просто определено, что эта функция руководствуется порядком сортировки, определенным операционной системой.

Пример

Вот как можно отсортировать массив строк в порядке, учитывающем регион:

```
var strings;  // Сортируемый массив строк; инициализируется в другом месте
strings.sort(function(a,b) { return a.localeCompare(b) });
```

String.match()

JavaScript 1.2; JScript 3.0; ECMAScript v3

находит одно или более соответствий регулярному выражению

Синтаксис

строка.match(*regexp*)

Аргументы

regexp

Объект RegExp, задающий маску для поиска. Если этот аргумент не является RegExp, он сначала преобразуется с помощью конструктора RegExp().

Возвращаемое значение

Массив, содержащий результаты поиска. Содержимое массива зависит от того, установлен ли в *regexp* глобальный атрибут g. Ниже это возвращаемое значение описано подробно.

Описание

match() ищет в *строке* соответствия маске, определенной выражением *regexp*. Поведение этого метода существенно зависит от того, установлен ли атрибут g в *regexp* или нет. Регулярные выражения подробно рассмотрены в главе 10.

Если в *regexp* нет атрибута g, match() ищет одно соответствие в *строке*. Если соответствие не найдено, match() возвращает null. В противном случае он возвращает массив, содержащий информацию о найденном соответствии. Элемент массива с индексом 0 содержит найденный текст. Оставшиеся элементы содержат текст, соответствующий всем подвыражениям внутри регулярного выражения. В дополнение к этим обычным элементам массива, возвращаемый массив имеет еще два объектных свойства. Свойство index массива указывает позицию начала найденного текста внутри *строки*. Кроме того, свойство input возвращаемого массива содержит ссылку на саму *строку*.

Если в *regexp* имеется флаг g, match() выполняет глобальный поиск, находя в *строке* все соответствующие подстроки. Он возвращает null, если соответствие не найдено, и возвращает массив, если найдено одно и более соответствий. Однако содержимое полученного массива при глобальном поиске существенно отличается. В этом случае элементы массива содержат все найденные подстроки в строке, а сам массив не имеет свойств index и input. Обратите внимание, что для глобального поиска match() не предоставляет информации о подвыражениях в скобках и не указывает, в каком месте *строки* было найдено каждое из соответствий. Эту информацию для глобального поиска можно получить посредством RegExp.exec().

Пример

Следующий глобальный поиск находит все числа в строке:

```
"1 plus 2 equals 3".match(/\d+/g)  // Возвращает ["1", "2", "3"]
```

Следующий неглобальный поиск выполняется при помощи более сложного регулярного выражения с несколькими подвыражениями. Оно находит URL, а его подвыражения соответствуют протоколу, хосту и пути в URL:

```
var url = /(\w+):\/\/([\w.]+)\/(\S*)/;
var text = "Visit my home page at http://www.isp.com/~david";
var result = text.match(url);
if (result != null) {
    var fullurl = result[0];   // Содержит "http://www.isp.com/~david"
    var protocol = result[1];  // Содержит "http"
    var host = result[2];      // Содержит "www.isp.com"
    var path = result[3];      // Содержит "~david"
}
```

Смотри также

RegExp, RegExp.exec(), RegExp.test(), String.replace(), String.search(); главу 10

String.replace() JavaScript 1.2; JScript 3.0; ECMAScript v3

заменяет подстроку (подстроки), соответствующие регулярному выражению

Синтаксис

строка.replace(*regexp*, *замена*)

Аргументы

regexp

Объект RegExp, задающий маску заменяемой подстроки. Если этот аргумент представляет собой строку, то он выступает в качестве текстового шаблона для поиска и не преобразуется в объект RegExp.

замена

Строка, задающая текст замены, или функция, вызываемая для генерации текста замены. Подробности см. ниже в разделе «Описание».

Возвращаемое значение

Новая строка, в которой первое или все соответствия регулярному выражению *regexp* заменены строкой *замена*.

Описание

replace() выполняет операцию поиска-замены со *строкой*. Он ищет в *строке* одну или несколько подстрок, соответствующих регулярному выражению *regexp*, и заменяет их значением аргумента *замена*. Если в *regexp* указан глобальный атрибут g, replace() заменяет все найденные подстроки. В противном случае он заменяет только первую найденную подстроку.

Параметр *замена* может быть либо строкой, либо функцией. Если это строка, то каждое найденное соответствие заменяется указанной строкой. Исключение, однако, составляет символ $, который имеет особый смысл в строке *замена*. Как показано в следующей таблице, он обозначает, что для замены используется строка, производная от найденного соответствия.

Символы	Замена
$1, $2, ... $99	Текст, соответствующий подвыражению с номером от 1 до 99 внутри регулярного выражения *regexp*
$&	Подстрока, соответствующая *regexp*
$`	Текст слева от найденной подстроки
$'	Текст справа от найденной подстроки
$$	Символ доллара

В ECMAScript v3 определено, что аргумент *замена* может быть функцией, а не строкой, и эта возможность реализована в JavaScript 1.2 и JScript 5.5. В этом случае функция вызывается для каждого найденного соответствия, а возвращаемая ею строка выступает в качестве текста для замены. Первый аргумент, передаваемый функции, представляет собой строку, соответствующую маске. Следующие за ним аргументы – это строки, соответствующие любым подвыражениям внутри маски. Таких аргументов может быть ноль или более. Следующий аргумент – это целое, указывающее позицию внутри *строки*, в которой было найдено соответствие, а последний аргумент функции *замена* – это сама *строка*.

Пример

Обеспечение правильного регистра букв в слове «JavaScript»:

```
text.replace(/javascript/i, "JavaScript");
```

Преобразование имени из формата «Doe, John» в формат «John Doe»:

```
name.replace(/(\w+)\s*,\s*(\w+)/, "$2 $1");
```

Замена всех двойных кавычек на две одинарные закрывающие и две одинарные открывающие кавычки:

```
text.replace(/"([^"]*)"/g, "``$1``");
```

Перевод первых букв всех слов в строке в верхний регистр:

```
text.replace(/\b\w+\b/g, function(word) {
                           return word.substring(0,1).toUpperCase() +
                                  word.substring(1);
                         });
```

Смотри также

RegExp, RegExp.exec(), RegExp.test(), String.match(), String.search(); главу 10

String.search()

JavaScript 1.2; JScript 3.0; ECMAScript v3

поиск соответствия регулярному выражению

Синтаксис

строка.search(*regexp*)

Аргументы

regexp

Объект RegExp, задающий маску, которая будет использоваться для поиска в *строке*. Если этот аргумент не является RegExp, он сначала преобразуется путем передачи его конструктору RegExp().

Возвращаемое значение

Позиция начала первой подстроки в *строке*, соответствующая *regexp*, или –1, если соответствие не найдено.

Описание

search() ищет подстроку в *строке*, соответствующую регулярному выражению *regexp*, и возвращает позицию первого символа найденной подстроки или –1, если соответствие не найдено.

search() не выполняет глобального поиска; он игнорирует флаг g. Он также игнорирует свойство *regexp*.lastIndex и всегда выполняет поиск с начала строки, следовательно, всегда возвращает позицию первого соответствия, найденного в *строке*.

Пример

```
var s = "JavaScript is fun";
s.search(/script/i)  // Возвращает 4
s.search(/a(.)a/)    // Возвращает 1
```

Смотри также

RegExp, RegExp.exec(), RegExp.test(), String.match(), String.replace(); главу 10

String.slice()

JavaScript 1.2; JScript 3.0; ECMAScript v3

извлечение подстроки

Синтаксис

строка.slice(*start*, *end*)

Аргументы

start

Индекс в строке, с которой должен начинаться фрагмент. Если этот аргумент отрицателен, он обозначает позицию, измеряемую от конца строки. То есть –1 обозначает последний символ, –2 – второй с конца и т. д.

end

Индекс символа исходной строки непосредственно после конца извлекаемого фрагмента. Если он не указан, фрагмент включает все символы от позиции *start* до конца строки. Если этот аргумент отрицателен, он обозначает позицию, отсчитываемую от конца строки.

Возвращаемое значение

Новая строка, которая содержит все символы *строки*, начиная с символа в позиции *start* (и включая его) и заканчивая символом в позиции *end* (но не включая его).

Описание

slice() возвращает строку, содержащую фрагмент, или подстроку *строки*. Он не изменяет *строку*.

Методы slice(), substring() и признанный устаревшим метод substr() объекта String возвращают части строки. slice() более гибок, чем substring(), поскольку допускает отрицательные значения аргументов. slice() отличается от substr() тем, что задает строку с помощью двух символьных позиций, а substr() использует одно значение позиции и длину. Заметьте также, что String.slice() является аналогом Array.slice().

Пример

```
var s = "abcdefg";
s.slice(0,4)    // Возвращает "abcd"
s.slice(2,4)    // Возвращает "cd"
s.slice(4)      // Возвращает "efg"
s.slice(3,-1)   // Возвращает "def"
s.slice(3,-2)   // Возвращает "de"
s.slice(-3,-1)  // Должен возвращать "ef"; в IE 4 возвращает "abcdef"
```

Ошибки

Отрицательные значения *start* не работают в JScript 3.0 (Internet Explorer 4). Они обозначают не символьную позицию, отсчитываемую от конца строки, а позицию 0.

Смотри также

Array.slice(), String.substring()

String.split() JavaScript 1.1; JScript 3.0; ECMAScript v1; расширено в ECMAScript v3

разбивают строку на массив строк

Синтаксис

строка.split(*разделитель*, *лимит*)

Аргументы

разделитель

Строка или регулярное выражение, по которому разбивается строка. Использование регулярного выражения в качестве разделителя стандартизовано в ECMAScript v3 и реализовано в JavaScript 1.2 и JScript 3.0; оно не реализовано в JavaScript 1.1.

лимит

Это необязательное целое задает максимальную длину полученного массива. Если этот аргумент указан, то количество возвращенных подстрок не будет превышать указанное. Если он не указан, разбивается вся строка независимо от ее длины. Этот аргумент стандартизован в ECMAScript v3 и реализован в JavaScript 1.2 и JScript 3.0; он не реализован в JavaScript 1.1.

Возвращаемое значение

Массив строк, создаваемый путем разбиения *строки* на подстроки по границам, заданным *разделителем*. Подстроки в возвращаемом массиве не включают сам *разделитель*, кроме случая, отмеченного ниже.

Описание

Метод split() создает и возвращает массив подстрок указанной строки, причем размер возвращаемого массива не превышает указанный *лимит*. Эти подстроки создаются путем поиска текста, соответствующего *разделителю*, в строке от начала до конца и разбиения строки до и после найденного текста. Ограничивающий текст не включается ни в одну из возвращаемых строк, кроме случая, отмеченного ниже. Заметьте, что если *разделитель* соответствует началу строки, первый элемент возвращаемого массива будет пустой строкой – текстом, присутствующим перед разделителем. Аналогично, если разделитель соответствует концу строки, последний элемент массива (если это не противоречит значению аргумента *лимит*) будет пустой строкой.

Если *разделитель* не указан, строка вообще не разбивается, и возвращаемый массив содержит только один строковый элемент, представляющий собой строку целиком. Если *разделитель* представляет собой пустую строку или регулярное выражение, соответствующее пустой строке, то строка разбивается между каждым символом, а возвращаемый массив имеет ту же длину, что и исходная строка, если не указан меньший *лимит*. (Заметьте, что это особый случай, поскольку пустая строка перед первым и после последнего символа отсутствует.)

Выше мы сказали, что подстроки в массиве, возвращаемом этим методом, не содержат текста разделителя, использованного для разбиения строки. Однако если *разделитель* – это регулярное выражение, содержащее подвыражения в скобках, то подстроки, соответствующие этим подвыражениям (но не текст, соответствующий регулярному выражению в целом), включаются в возвращаемый массив.

Метод String.split() – обратный для метода Array.join().

Пример

Метод split() наиболее полезен при работе с сильно структурированными строками:

```
"1:2:3:4:5".split(":");  // Возвращает ["1","2","3","4","5"]
"|a|b|c|".split("|");    // Возвращает ["", "a", "b", "c", ""]
```

Еще одно распространенное применение метода split() состоит в разборе команд и других подобных строк для разбиения их на слова, разделенные пробелами:

```
var words = sentence.split(' ');
```

В разделе «Ошибки» подробно рассмотрен особый случай, когда в качестве *разделителя* выступает один пробел. Проще разбить строку на слова, используя в качестве разделителя регулярное выражение:

```
var words = sentence.split(/\s+/);
```

Для того чтобы разбить строку на массив символов, возьмите в качестве разделителя пустую строку. Если требуется разбить на массив символов только часть строки, задайте аргумент *лимит*:

```
"hello".split("");     // Возвращает ["h","e","l","l","o"]
"hello".split("", 3);  // Возвращает ["h","e","l"]
```

Если необходимо, чтобы разделители или одна и более частей разделителя были включены в результирующий массив, напишите регулярное выражение с подвыражениями в скобках. Так, следующий код разбивает строку по тегам HTML и включает эти теги в результирующий массив:

```
var text = "hello <b>world</b>";
text.split(/(<[^>]*>)/);  // Возвращает ["hello ","<b>","world","</b>",""]
```

Ошибки

В реализациях JavaScript от Netscape, когда явно указывается версия 1.2 (например, с помощью атрибута language тега <script>), метод split() в ведет себя особым образом: если в качестве *разделителя* определен один пробел, то метод разбивает строку по пробелам, но игнорирует любые символы-разделители в начале и в конце строки. Подробнее см. в разделе 11.6 «Несовместимость Netscape JavaScript 1.2».

Смотри также

Array.join(), RegExp;

String.substr()

JavaScript 1.2; JScript 3.0; признан устаревшим

извлекает подстроку

Синтаксис

строка.substr(*начало*, *длина*)

Аргументы

начало

Начальная позиция подстроки. Если аргумент отрицателен, он обозначает позицию, измеряемую от конца строки: –1 обозначает последний символ, –2 – второй символ с конца и т. д.

длина

Количество символов в подстроке. Если этот аргумент отсутствует, возвращаемая строка включает все символы от начальной позиции до конца строки.

Возвращаемое значение

Копия фрагмента *строки*, начиная с символа, находящегося в позиции *начало* (включительно) и имеющая длину, равную аргументу *длина*, или заканчивающаяся концом строки, если *длина* не указана.

Описание

`substr()` извлекает и возвращает подстроку *строки*. Он не изменяет *строку*.

Обратите внимание, что `substr()` задает нужную подстроку с помощью символьной позиции и длины. Благодаря этому появляется полезная альтернатива методам `String.substring()` и `String.splice()`, в которых подстрока задается двумя символьными позициями. Заметьте, однако, что этот метод не был стандартизован в ECMAScript и, следовательно, считается устаревшим.

Пример

```
var s = "abcdefg";
s.substr(2,2);   // Возвращает "cd"
s.substr(3);     // Возвращает "defg"
s.substr(-3,2);  // Должен возвращать "ef"; в IE 4 возвращает "ab"
```

Ошибки

Отрицательные значения аргумента *начало* не работают в JScript 3.0 (IE 4). Они задают не позицию символа, отсчитываемую от конца строки, а позицию 0.

Смотри также

String.slice(), String.substring()

String.substring() JavaScript 1.0; JScript 1.0, ECMAScript v1

возвращает подстроку строки

Синтаксис

строка`.substring(`*от*`, `*до*`)`

Аргументы

от Целое, задающее позицию первого символа требуемой подстроки внутри *строки*.

до Необязательное целое, на единицу большее позиции последнего символа требуемой подстроки. Если этот аргумент опущен, возвращаемая подстрока заканчивается в конце *строки*.

Возвращаемое значение

Новая строка длиной *до* – *от*, содержащая подстроку из *строки*. Новая строка содержит символы, скопированные с позиций в *строке*, начиная с *от* и заканчивая *до* –1.

Описание

`String.substring()` возвращает подстроку *строки*, содержащую символы между позициями *от* и *до*. Символ в позиции *от* включается в подстроку, а символ в позиции *до* не включается.

Если *от* равно *до*, этот метод возвращает пустую строку (длиной 0). Если *от* больше, чем *до*, этот метод сначала меняет два аргумента местами, а затем возвращает строку между ними.

Помните, что символ в позиции *от* включается в подстроку, а символ в позиции *до* в нее не включается. Может показаться, что это взято с потолка и нелогично, но особенность такой системы в том, что длина возвращаемой подстроки всегда равна *до*–*от*.

Обратите внимание, что для извлечения подстрок из строки также могут применяться `String.slice()` и нестандартный метод `String.substr()`.

Ошибки

В реализациях JavaScript от Netscape, когда явно запрашивается использование версии 1.2 (например, с помощью атрибута `language` тега `<script>`), этот метод не выполняет корректную перестановку своих аргументов, если *от* больше, чем *до*. Вместо этого метод возвращает пустую строку.

Смотри также

String.charAt(), String.indexOf(), String.lastIndexOf(), String.slice(), String.substr()

String.toLocaleLowerCase() JavaScript 1.5; JScript 5.5, ECMAScript v3

преобразует строку в нижний регистр

Синтаксис

строка.`toLocaleLowerCase()`

Возвращаемое значение

Копия *строки*, преобразованная в нижний регистр специфическим для региона образом. Только немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что метод `toLowerCase()`.

Смотри также

String.toLocaleUpperCase(), String.toLowerCase(), String.toUpperCase()

String.toLocaleUpperCase() JavaScript 1.5; JScript 5.5, ECMAScript v3

преобразует строку в верхний регистр

Синтаксис

строка.`toLocaleUpperCase()`

Возвращаемое значение

Копия *строки*, преобразованная в верхний регистр специфическим для региона образом. Лишь немногие языки, такие как турецкий, имеют специфические для региона соответствия регистров, поэтому данный метод обычно возвращает то же значение, что метод `toUpperCase()`.

Смотри также

String.toLocaleLowerCase(), String.toLowerCase(), String.toUpperCase()

String.toLowerCase()

JavaScript 1.0; JScript 1.0, ECMAScript v1

преобразует строку в нижний регистр

Синтаксис

строка.toLowerCase()

Возвращаемое значение

Копия *строки*, в которой все буквы верхнего регистра преобразованы в эквивалентные им буквы нижнего регистра, если такие имеются.

String.toString()

JavaScript 1.0; JScript 1.0, ECMAScript v1

возвращает строку **перекрывает Object.toString()**

Синтаксис

строка.toString()

Возвращаемое значение

Элементарное строковое значение *строки*. Вызов этого метода редко бывает необходим.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося объектом String.

Смотри также

String.valueOf()

String.toUpperCase()

JavaScript 1.0; JScript 1.0, ECMAScript v1

преобразует строку в верхний регистр

Синтаксис

строка.toUpperCase()

Возвращаемое значение

Копия *строки*, в которой все буквы нижнего регистра преобразованы в эквивалентные им буквы верхнего регистра, если такие имеются.

String.valueOf()

JavaScript 1.0; JScript 1.0, ECMAScript v1

возвращает строку **перекрывает Object.valueOf()**

Синтаксис

строка.valueOf()

Возвращаемое значение

Элементарное строковое значение *строки*.

Исключения

TypeError

Если метод вызывается для объекта, не являющегося объектом String.

См. также

String.toString()

SyntaxError

JavaScript 1.5; JScript 5.5; ECMAScript v3

сигнализирует о синтаксической ошибке **наследуется от Error**

Конструктор

`new SyntaxError()`

`new SyntaxError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта SyntaxError.

Возвращаемое значение

Только что созданный объект SyntaxError. Если аргумент *сообщение* указан, объект SyntaxError возьмет его в качестве значения своего свойства `message`; в противном случае он возьмет в качестве значения этого свойства строку по умолчанию, определенную в реализации. Конструктор `SyntaxError()`, вызванный как функция, без оператора `new`, ведет себя точно так же, как если бы он был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробнее см. в стстье «Error.message».

`name`

Строка, задающая тип исключения. Все объекты SyntaxError наследуют для этого свойства значение «SyntaxError».

Описание

Экземпляр класса SyntaxError сигнализирует о синтаксической ошибке в коде JavaScript. Метод `eval()`, конструктор `Function()` и конструктор `RegExp()` могут генерировать исключения этого типа. Подробности о генерации и перехвате исключений см. в статье «Error».

Смотри также

Error, Error.message, Error.name

TypeError

JavaScript 1.5; JScript 5.5; ECMAScript v3

генерируется, когда значение имеет неверный тип **наследуется от Error**

Конструктор

`new TypeError()`

`new TypeError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта TypeError.

Возвращаемое значение

Только что созданный объект TypeError. Если аргумент *сообщение* указан, объект TypeError возьмет его в качестве значения своего свойства `message`; в противном случае он возьмет в качестве значения этого свойства строку по умолчанию, определенную в реализации. Конструктор `TypeError()`, вызванный как функция, без оператора `new`, ведет себя точно так же, как если бы он был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, содержащее дополнительную информацию об исключении. Это свойство создержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробности см. в статье «Error.message».

`name`

Строка, задающая тип исключения. Все объекты TypeError наследуют для этого свойства значение «TypeError».

Описание

Экземпляр класса TypeError создается, когда значение имеет не тот тип, который ожидается. Это чаще всего происходит при попытке обратиться к свойству `null` или к неопределенному объектному значению. Это исключение может также возникнуть, если вызван метод, определенный одним классом, для объекта, являющегося экземпляром какого-либо другого класса, или если оператор `new` применяется к значению, не являющемуся функцией-конструктором. Реализациям JavaScript также разрешено создавать объекты `TypeError`, когда встроенная функция или метод вызывается с большим числом аргументов, чем ожидается. Генерация и перехват исключений подробно рассмотрены в статье «Error».

Смотри также

Error, Error.message, Error.name

undefined

JavaScript 1.5; JScript 5.5; ECMAScript v3

неопределенное значение

Синтаксис

undefined

Описание

undefined — это глобальное свойство, хранящее значение undefined JavaScript. Это то же самое значение, которое возвращается, когда вы пытаетесь прочитать значение несуществующего свойства объекта. Свойство undefined не перечисляется циклами for/in и не может быть удалено оператором delete. Заметьте, что undefined не является константой и может быть установлено равным любому другому значению, но лучше этого не делать.

Для проверки неопределенности значения применяйте оператор ===, поскольку оператор == рассматривает undefined как равное null.

unescape()

JavaScript 1.0; JScript 1.0; ECMAScript v1; устарело в ECMAScript v3

декодирует строку с управляющими последовательностями

Синтаксис

unescape(*s*)

Аргументы

s — строка, подлежащая декодированию.

Возвращаемое значение

Декодированная копия *s*.

Описание

unescape() — это глобальная функция, декодирующая строку, закодированную с помощью escape(). Она декодирует строку *s* путем поиска и замены последовательности символов в формате %*xx* и %u*xxxx* (где *x* — шестнадцатеричная цифра) на символы Unicode \u00*xx* и \u*xxxx*.

Несмотря на то что unescape() был стандартизован в первой версии ECMAScript, этот метод был признан устаревшим и исключен из стандарта в спецификации ECMAScript v3. Реализации ECMAScript могут реализовывать эту функцию, но это не обязательное требование. В JavaScript 1.5, а также в JScript 5.5 и более поздних версиях следует применять decodeURI() и decodeURIComponent(), а не unescape(). Подробности и пример см. в статье «escape()».

Смотри также

decodeURI(), decodeURIComponent(), escape(), String

URIError

JavaScript 1.5; JScript 5.5; ECMAScript v3

генерируется методами кодирования и декодирования URI **наследуется от Error**

Конструктор

`new URIError()`

`new URIError(`*сообщение*`)`

Аргументы

сообщение

Необязательное сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Если этот аргумент указан, он выступает в качестве значения свойства `message` объекта URIError.

Возвращаемое значение

Только что созданный объект URIError. Если аргумент *сообщение* указан, объект URIError возьмет его в качестве значения своего свойства `message`; в противном случае он возьмет в качестве значения этого свойства строку по умолчанию, определенную в реализации. Конструктор `URIError()`, вызванный как функция, без оператора `new`, ведет себя точно так же, как если бы он был вызван с оператором `new`.

Свойства

`message`

Сообщение об ошибке, предоставляющее дополнительную информацию об исключении. Это свойство содержит строку, переданную конструктору, или строку по умолчанию, определенную в реализации. Подробнее см. в статье «Error.message».

`name`

Строка, задающая тип исключения. Все объекты URIError наследуют для этого свойства значение «URIError».

Описание

Экземпляр класса URIError создается функциями `decodeURI()` и `decodeURIComponent()`, если указанная строка содержит недопустимые шестнадцатеричные управляющие последовательности. Это исключение может генерироваться методами `encodeURI()` и `encodeURIComponent()`, если указанная строка содержит недопустимые пары Unicode. Генерация и перехват исключений подробно рассмотрены в статье «Error».

Смотри также

Error, Error.message, Error.name

Часть IV. Справочник по клиентскому JavaScript

Данная часть книги представляет собой полный справочник по всем объектам, свойствам, функциям, методам и обработчикам событий клиентского JavaScript. На первых страницах поясняется, как работать с этим справочным материалом.

Справочник по клиентскому JavaScript

В этом справочнике описаны классы, методы, свойства и обработчики событий, которые определены в веб-броузерах, поддерживающих клиентский JavaScript. Эти классы, методы и свойства образуют стандарт «де-факто» под названием DOM Level 0 API. Начинающим программистам, желающим иметь обратную совместимость, целесообразно использовать этот раздел совместно с частью III «Справочник по базовому JavaScript». Вводная часть и образец справочной статьи призваны помочь вам понять, как работать со справочником и извлечь из него максимум пользы. Не пожалейте времени на внимательное прочтение этого материала, и вам будет легче искать и использовать нужную информацию!

Материал справочника организован в алфавитном порядке. Справочные страницы методов и свойств классов отсортированы по их полным именам, включающим имена классов, в которых они определены. Например, для того чтобы найти метод `submit()` класса Form, надо искать «Form.submit», а не «submit».

С целью экономии места в этом расширенном четвертом издании книги большинство свойств в этом справочном разделе не имеют отдельных справочных страниц (однако для всех методов и обработчиков событий есть свои справочные страницы). А простые свойства полностью описаны на справочной странице для определяющего их класса. Так, свойство `images[]` класса Document рассмотрено на справочной странице «Document». Нетривиальные свойства, требующие обстоятельного пояснения, имеют собственные справочные страницы, ссылки на которые можно найти на справочной странице класса или интерфейса, определяющего эти свойства. Например, разыскивая свойство `cookie` на справочной странице «Document» или свойство `status` на справочной странице «Window», вы найдете краткое описание свойства и ссылку на страницы под заголовками «Document.cookie» и «Window.status».

В клиентском JavaScript имеется несколько глобальных свойств и функций, таких как `window`, `history` и `alert()`. В клиентском JavaScript в качестве глобального объекта выстпает объект Window, а «глобальные» свойства и функции клиентского JavaScript фактически представляют собой свойства класса Window. Поэтому в данном справочнике глобальные свойства и функции описаны на справочной странице «Window» или под такими именами, как «Window.alert()».

Может оказаться, что вы не помните имени класса или интерфейса, определяющего метод или свойство, которое ищете, или не знаете точно, в какой справочной статье искать данный класс или интерфейс. Часть VI этой книги представляет собой специальный указатель, предназначенный для оказания помощи в таких ситуациях. Допустим, надо найти «Button». В указателе вы увидите, что класс Button описан в «Справочнике по клиентскому JavaScript». А если вы ищете имя «alert», то в части VI узнаете, что `alert()` – это метод клиентского класса Window.

Найдя справочную страницу, вы без труда отыщете на ней нужную информацию. Но вам будет легче работать с этим справочником, если вы поймете, как написаны и организованы справочные статьи. Далее показана структура всех справочных статей и рассказано, какую информацию можно в этих статьях найти. Прежде чем искать что-то в справочнике, не пожалейте времени на ознакомление с этим примером.

Пример справочной статьи

доступность

как читать справочные статьи по клиентскому JavaScript **наследование**

Заголовок и краткое описание

Любая справочная статья начинается с блока заголовка, состоящего из четырех частей. Статьи отсортированы по заголовку. Краткое описание поможет вам быстро решить, интересует ли вас остальная часть страницы.

Доступность

Информация о доступности приводится в правом верхнем углу блока заголовка. Здесь говорится, когда был введен этот класс, метод или обработчик событий. Для некоторых менее переносимых элементов клиентского JavaScript в этом разделе указано, какие версии Netscape и Internet Explorer их поддерживают. Если предмет статьи хорошо поддерживается веб-броузерами, и поддержка была добавлена в Netscape и IE в одном поколении броузеров, в этом разделе указана доступность в терминах версии базового JavaScript. Конкретные соответствующие версии Netscape и Internet Explorer можно определить по таблицам в главе 1. Естественно, информация о доступности свойств, не имеющих собственных справочных страниц, отсутствует. Однако если доступность свойства отличается от доступности определяющего его класса, этот факт отмечается в описании свойства.

Наследование

Если класс является наследником суперкласса, об этом говорится в нижнем правом углу блока заголовка. Классы JavaScript могут наследовать свойства и методы от других классов (см. главу 8). Например, класс Button наследует от Input, который, в свою очередь, является наследником HTMLElement. Благодаря информации о наследовании мы знаем, какие суперклассы имеет смысл просмотреть.

Синтаксис

На каждой справочной странице есть раздел «Синтаксис», описывающий применение класса, метода или обработчика событий в программе. Пример для класса Form:

```
document.имя_формы
document.forms[номер_формы]
```

Здесь показано два различных способа обращения к объекту Form. Текст, написанный моноширинным шрифтом, должен быть набран без изменений. *Моноширинный курсив* означает, что текст должен быть заменен на что-то другое. *имя_формы* должно быть заменено на реальное имя формы, а *номер_формы* – на индекс формы в массиве forms[]. Аналогично *document* в этом синтаксисе должен быть заменен ссылкой на объект Document. Обратившись к разделу «Синтаксис» справочной страницы «Document», мы выясним, что для Document имеется также две формы:

```
document
window.document
```

Другими словами, можно заменить *document* на идентификатор document или на *window*.document. В последнем случае надо будет ознакомиться с синтаксисом класса Window, чтобы выяснить, как обращаться к объекту Window, – то есть на что можно заменить *window*.

Аргументы

Если справочная страница описывает метод, то за разделом «Синтаксис» следует подраздел «Аргументы», в котором описаны аргументы метода. Если метод не имеет аргументов, этот подраздел отсутствует.

аргумент1

В этом списке описываются аргументы, например описание аргумента *аргумент1*.

аргумент2

А это описание аргумента *аргумент2*.

Возвращаемое значение

В этом разделе описано возвращаемое методом значение. Если метод не возвращает значения, этот подраздел опускается.

Конструктор

Если справочная страница описывает класс, у которого имеется метод-конструктор, то в этом разделе показано, как с помощью метода-конструктора создаются экземпляры класса. Конструкторы также являются методами, поэтому раздел «Конструктор» выглядит во многом так же, как раздел «Синтаксис» справочной страницы метода и также имеет подраздел «Аргументы».

Свойства

Если справочная страница описывает класс, то в разделе «Свойства» перечислены и описаны свойства, определяемые в этом классе. В данном справочнике собственные справочные страницы есть только у особенно сложных свойств.

свойство1

Это описание свойства свойство1, включая его тип, назначение (смысл) и доступность (только для чтения или для чтения и записи).

свойство2

То же самое для свойства свойство2.

Методы

Справочная страница для класса, определяющего методы, включает раздел «Методы», в котором перечислены имена методов и предоставляется краткое описание для каждого из них. Полное описание каждого метода находится на отдельной справочной странице.

Обработчик событий

Справочная страница для класса, определяющего обработчики событий, включает раздел «Обработчики событий», в котором перечислены имена обработчиков и краткое описание каждого из них. Полное описание каждого обработчика события приводится на отдельной справочной странице.

Синтаксис HTML

Многие классы клиентского JavaScript имеют свои аналоги в HTML. Эта справочная страница включает раздел, в котором приведен синтаксис HTML с комментариями, используемый для создания элемента HTML, соответствующего объекту JavaScript.

Описание

Большинство справочных страниц содержат раздел «Описание», являющийся базовым описанием класса, метода или обработчика события, которому посвящена эта страница. Это сердце справочной страницы. При первом знакомстве с классом, методом или обработчиком можно перейти непосредственно к этому разделу, а затем вернуться обратно к предыдущим, таким как «Аргументы», «Свойства» и «Методы». Если же предмет данной справочной статьи для вас не нов, то не исключено, что читать этот раздел не обязательно, и в поисках конкретной информации вы заглянете, например, в раздел «Аргументы» или «Свойства».

Некоторые статьи этого справочника представляют собой не более чем короткий абзац. Другие могут занимать одну или несколько страниц. Некоторые простые методы исчерпываются разделами «Аргументы» и «Возвращаемое значение», поэтому раздел «Описание» в них отсутствует.

Пример

Некоторые страницы содержат пример, иллюстрирующий типичное применение предмета статьи. Большинство страниц, однако, не содержат примеров, – их можно найти в первой половине книги.

Ошибки

Если предмет справочной статьи работает не совсем правильно, то в данном разделе приведено описание ошибок. Однако заметим, что в данной книге не ставилась задача учесть все ошибки во всех версиях и реализациях клиентского JavaScript.

См. также

Большинство справочных страниц завершаются перекрестными ссылками на связанные справочные страницы, которые могут представлять для вас интерес. Большинство перекрестных ссылок относятся к другим справочным страницам в этом справочнике. Однако некоторые ссылаются на индивидуальные описания свойств, содержащиеся на справочной странице класса, а другие – на связанные справочные страницы в «Справочнике по W3C DOM» (часть V) или на главы в двух первых частях книги.

Anchor

JavaScript 1.2

цель гиперссылки **наследуется от HTMLElement**

Синтаксис

document.anchors[*i*]
document.anchors.length

Свойства

Anchor наследует свойства от HTMLElement и определяет или перекрывает следующие свойства:

name

Содержит имя объекта Anchor. Значение этого свойства изначально устанавливается равным атрибуту name тега <a>.

text *[Netscape 4]*

Это свойство задает обычный текст, который может находиться между тегами <a> и якорного элемента. Обратите внимание, что это свойство корректно работает, только если между тегами <a> и нет промежуточных тегов. Если присутствуют другие теги HTML, свойство text может содержать только часть текста якоря.

HTMLElement.innerText предоставляет эквивалент в IE 4 этого специфического для Netscape свойства.

Синтаксис HTML

Объект Anchor создается любым стандартным тегом HTML <a>, в котором установлен атрибут name:

```
<a
  name="name"  // Ссылки могут обращаться к этому якорю по его имени
>
текст
</a>
```

Описание

Якорный элемент (якорь) – это именованный участок внутри документа HTML. Якоря создаются с помощью тега <a>, в котором указан атрибут name. Объект Document имеет свойство-массив anchors[], содержащий объекты Anchor, которые представляют все якорные элементы в документе. Этот массив anchors[] существовал еще в JavaScript 1.0, но объект Anchor не был реализован вплоть до JavaScript 1.2. Поэтому элементы anchors[] были равны null вплоть до JavaScript 1.2.

Обратите внимание, что тег <a>, посредством которого создаются якорные элементы, также служит для создания гиперссылок. Несмотря на то что в терминологии HTML гиперссылки часто называются якорями, в JavaScript они представляются объектами Link, а не объектами Anchor. Однако далее в этой книге, в «Справочнике по W3C DOM», и якоря и ссылки описываются в статье «HTMLAnchorElement».

См. также

Свойство anchors[] объекта Document, Link; HTMLAnchorElement в «Справочнике по W3C DOM»

Applet

JavaScript 1.1

апплет, встроенный в веб-страницу

Синтаксис

document.applets[*i*]
document.*имя_апплета*

Свойства

Свойства объекта Applet совпадают с открытыми полями представляемого им Java-апплета.

Методы

Методы объекта Applet совпадают с открытыми методами представляемого им Java-апплета.

Описание

Объект Applet представляет Java-апплет, встроенный в документ HTML. Свойства объекта Applet представляют открытые поля апплета, а методы объекта Applet – открытые методы апплета. Благодаря технологиям LiveConnect в Netscape и ActiveX в Internet Explorer программы JavaScript могут использовать объект Applet для чтения и записи полей и вызова методов соответствующего Java-апплета. Подробности см. в главе 22.

Помните, что Java – это строго типизированный язык. Это значит, что все поля апплета объявляются с определенным типом данных, и присваивание им значения какого-либо другого типа приводит к ошибке времени выполнения. То же самое относится и к методам апплета: каждый аргумент имеет определенный тип, и аргументы не могут быть опущены, как это можно делать в JavaScript.

См. также

JavaObject; главу 22

Area

см. Link

Button

JavaScript 1.0; расширен в JavaScript 1.1

графическая кнопка **наследуется от Input, HTMLElement**

Синтаксис

form.*имя_кнопки*
form.elements[*i*]

Свойства

Button наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

value

Строковое свойство, задающее текст, помещенный на кнопку. Его значение указывается в атрибуте value тега HTML <input>, который создал кнопку. В броузе-

рах, которые не могут переформатировать содержимое документа, это свойство может быть доступно только для чтения.

Методы

Button наследует методы от Input и HTMLElement.

Обработчик событий

Button наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующие:

`onclick`

Вызывается при щелчке по кнопке.

Синтаксис HTML

Элемент Button создается с помощью стандартного тега HTML `<input>`:

```
<form>
    ...
  <input
    type="button"              // Указывает, что это кнопка
    value="метка"              // Текст, который должен присутствовать внутри кнопки,
                               // задается свойством value
    [ name="имя" ]             // Имя, по которому позже можно обращаться к кнопке
                               // Задает значение свойства name
    [ onclick="обработчик" ]   // Инструкции JavaScript, исполняемые при щелчке по кнопке
  >
    ...
</form>
```

Объекты Button также могут создаваться с помощью тегов HTML 4 `<button>`:

```
<button id="имя"
        onclick="обработчик">
метка
</button>
```

Описание

Элемент Button представляет графическую кнопку в форме внутри документа HTML. Свойство `value` содержит текст, отображаемый кнопкой. Свойство `name` — это имя кнопки, по которому к ней можно обращаться. Обработчик события `onclick` вызывается, когда пользователь щелкает по кнопке.

Применение

Элемент Button применяется при необходимости обеспечить пользователю возможность вызова некоторого действия на вашей странице. Иногда для той же цели годится объект Link, но если требуемым значением не является переход по ссылке, следует предпочесть Button, так как этот элемент более явно указывает пользователю, что должно быть выполнено некоторое действие.

Обратите внимание, что элементы Submit и Reset являются видами кнопок, которые передают данные формы или очищают ее значения. Часто этих действий по умолчанию для формы достаточно и создавать какие-либо другие виды кнопок не требуется.

Пример

```
<form name="form1">
    <input type="button"
        name="press_me_button"
        value="Нажми меня"
        onclick="username = prompt('Назовите ваше имя','')"
    >
</form>
```

См. также

Form, HTMLElement, Input, Reset, Submit; HTMLInputElement в «Справочнике по W3C DOM»

Button.onclick

JavaScript 1.0

обработчик, вызываемый при щелчке по кнопке Button

Синтаксис

```
<input type="button" value="текст-кнопки" onclick="обработчик">
button.onclick
```

Описание

Свойство `onclick` объекта Button ссылается на функцию обработки события, которая вызывается при щелчке по кнопке. Подробности см. в статье «HTMLElement.onclick». Заметьте, однако, что `Button.onclick` поддерживается начиная с JavaScript 1.0, в отличие от обобщенного обработчика `HTMLElement.onclick`.

См. также

HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Checkbox

JavaScript 1.0; расширен в JavaScript 1.1

флажок **наследуется от Input, HTMLElement**

Синтаксис

К одиночному элементу Checkbox с уникальным именем можно обращаться любым из этих способов:

```
form.checkbox_name
form.elements[i]
```

Когда форма содержит группу флажков с одинаковыми именами, они помещаются в массив, и к ним можно обращаться следующим образом:

```
form.checkbox_name[j]
form.checkbox_name.length
```

Свойства

Checkbox наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

`checked`

Логическое свойство, доступное для чтения и для записи. Если флажок установлен, свойство `checked` равно `true`. Если флажок сброшен, `checked` равно `false`.

Если checked установлено в true, флажок выглядит как отмеченный. Аналогично, если это свойство установлено в false, флажок выглядит неотмеченным. Заметьте, что установка свойства checked не приводит к вызову обработчика события onclick метода Checkbox.

defaultChecked

Логическое свойство, доступное только для чтения, которое задает начальное состояние флажка. Равно true, если флажок был изначально отмечен. Другими словами, если в теге HTML <input> для флажка присутствовал атрибут checked. Если этот атрибут отсутствует, флажок изначально не отмечен и defaultChecked равен false.

value

Строковое свойство, доступное для чтения и записи, задающее текст, передаваемый на веб-сервер, если при передаче данных формы флажок установлен. Начальное значение value задается атрибутом value тега HTML <input> для флажка. Если атрибут value не указан, строка value по умолчанию равна «on».

Заметьте, что поле value не указывает, выбран ли флажок; текущее состояние флажка задает свойство checked. Когда в форме, отсылаемой на сервер, определяется группа связанных флажков, совместно использующих одно общее имя, важно, чтобы для каждого из них было задано уникальное значение атрибута value.

Методы

Checkbox наследует методы Input и HTMLElement.

Обработчик событий

Checkbox наследует обработчики событий от Input и HTMLElement и определяет или перекрывает обработчик:

onclick

Вызывается при щелчке по флажку.

Синтаксис HTML

Элемент Checkbox создается с помощью стандартного тега HTML <input> . Несколько элементов Checkbox часто создаются в группах путем указания нескольких тегов <input>, имеющих одинаковые атрибуты name.

```
<form>
    ...
  <input
    type="checkbox"            // Указывает, что это флажок
    [ name="имя" ]             // Имя, которое впоследствии позволит обратиться к флажку
                               // или сгруппировать несколько флажков с одним именем
                               // Задает значение свойства name
    [ value="значение" ]       // Значение, возвращаемое, когда этот флажок выбран
                               // Задает свойство value
    [ checked ]                // Указывает, что флажок изначально установлен
                               // Задает свойство defaultChecked
    [ onclick="обработчик" ]   // Инструкции JavaScript, исполняемые при щелчке по флажку
  >
метка                          // HTML-текст, который должен присутствовать рядом с флажком
    ...
</form>
```

Описание

Элементы Checkbox представляют графический флажок на форме HTML. Заметьте, что текст рядом с флажком не является частью самого элемента Checkbox и должен быть задан вне тега HTML `<input>`, описывающего данный флажок.

Обработчик события `onclick` позволяет задавать код JavaScript, исполняемый при установке и снятии флажка. Свойство `checked` можно проверить и определить состояние флажка, а можно задать его значение для установки или снятия флажка. Заметьте, что установка свойства `checked` изменяет графическое представление флажка, но не вызывает обработчик события `onclick`.

Хороший стиль программирования предусматривает задание атрибута `name` для флажка. Это обязательно, если флажок представляет собой часть формы, которая отсылает данные работающему на веб-сервере сценарию CGI. Задание атрибута `name` устанавливает свойство `name` и позволяет программисту обращаться в коде JavaScript к флажку по имени (а не как к члену массива `elements` формы), что повышает модульность и переносимость кода.

Например, если атрибут `name` флажка в форме `f` равен «opts», `f.opts` будет ссылаться на элемент Checkbox. Однако элементы Checkbox часто используются в связанных группах, и все элементы группы имеют одинаковые атрибуты `name` (общее имя, определяющее членов группы). В этом случае JavaScript помещает все элементы Checkbox группы в массив, а массив получает общее имя этих элементов. Если, например, все флажки формы `f`, относящиеся к группе, имеют атрибут `name`, установленный в «opts», `f.opts` является массивом элементов Checkbox, а `f.opts.length` — количеством элементов в массиве.

Можно установить атрибут `value` или свойство `value` флажка для задания строки, передаваемой на сервер при отправке данных формы, если флажок выбран. Когда флажок используется в одиночку, обычно бывает достаточно значения `value` по умолчанию, равного «on». Если же мы имеем дело с несколькими флажками, имеющими одинаковые имена, то для каждого из них должно быть указано отдельное значение, чтобы на сервер мог быть передан список значений выбранных флажков.

Применение

Элементы Checkbox могут предоставлять пользователю возможность выбора одного или нескольких вариантов. Этот тип элемента подходит для создания не взаимоисключающих вариантов выбора. Для взаимоисключающих списков опций предназначен элемент Radio.

См. также

Form, HTMLElement, Input, Radio; HTMLInputElement в «Справочнике по W3C DOM»

Checkbox.onclick

JavaScript 1.0

обработчик, вызываемый при щелчке по флажку

Синтаксис

```
<input type="checkbox" onclick="обработчик">
checkbox.onclick
```

Описание

Свойство onclick объекта Checkbox ссылается на функцию-обработчик событий, вызываемую при щелчке пользователем по флажку. Подробности см. в статье «HTMLElement.onclick». Однако обратите внимание, что Checkbox.onclick поддерживается начиная с JavaScript 1.0, в отличие от обобщенного обработчика HTMLElement.onclick.

См. также

HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Document

JavaScript 1.0; расширен в JavaScript 1.1, Netscape 4 и IE 4

представляет документ HTML **наследуется от HTMLElement**

Синтаксис

window.document
document

Свойства

Document наследует свойства от HTMLElement и определяет следующие свойства. Netscape и Internet Explorer определяют несколько несовместимых свойств объекта Document, применяемых в основном в DHTML; они перечислены по отдельности после этого списка.

alinkColor

alinkColor – это строковое свойство, задающее цвет активизированных ссылок в *document*. В броузерах этот цвет может устанавливаться между моментами, когда пользователь нажимает и отпускает кнопку мыши над ссылкой. Атрибут alink тега HTML <body> задает первоначальное значение этого свойства. Это свойство может быть установлено, но только в заголовке (<head>) документа. См. также цветовые свойства объекта HTMLBodyElement в «Справочнике по W3C DOM».

anchors[]

Массив объектов Anchor, по одному на каждый якорный элемент, присутствующих в *document*. Якорный элемент – это именованная позиция внутри документа, которая может служить целью гиперссылки. Массив anchors[] содержит anchors.length элементов, нумеруемых от нуля до anchors.length–1. Не путайте якоря с гиперссылками, которые представляются в JavaScript объектами Link в массиве Document.links[].

До JavaScript 1.2 объект Anchor не был реализован, и все элементы массива anchors[] были равны null.

applets[] *[JavaScript 1.1]*

Массив объектов Applet, по одному на каждый апплет, присутствующий в документе. При помощи объекта Applet можно читать и записывать все открытые переменные апплета и вызывать все открытые методы апплета. Если в теге <applet> имеется атрибут name, к апплету можно обратиться, приняв значение этого атрибута в качестве имени свойства document или в качестве индекса массива applets. Следовательно, если атрибут name первого апплета в документе равен «animator», то к нему можно обращаться любым из следующих способов:

```
document.applets[0]
document.animator
document.applets["animator"]
```

bgColor

Строковое свойство, задающее цвет фона в *document*. Первоначальное значение этого свойства задано в атрибуте bgcolor тега <body>. Цвет фона может быть изменен путем присваивания значения свойству bgColor. В отличие от других цветовых свойств, bgColor может быть установлено в любое время. См. также цветовые свойства объекта HTMLBodyElement в «Справочнике по W3C DOM».

cookie

Строка, являющаяся значением cookie, связанного с данным документом. См. справочную страницу «Document.cookie».

domain

Строка, задающая интернет-домен документа. Обеспечивает механизм безопасности. JavaScript 1.1 и более поздние версии. См. справочную страницу «Document.domain».

embeds[] *[JavaScript 1.1]*

Массив объектов, представляющих данные, встроенные в документ с помощью тега <embed>. Объекты в массиве embeds[] не ссылаются непосредственно на встраиваемые данные, а вместо этого ссылаются на объект, отображающий эти данные. С помощью объектов массива embeds[] можно организовать взаимодействие со встраиваемыми данными. Однако способ этого взаимодействия зависит от типа встраиваемых данных и подключаемого модуля (plug-in) или управляющего элемента ActiveX, применяемого для отображения данных. Чтобы узнать, может ли этот объект использоваться в сценариях JavaScript, и если да, то какой API им поддерживается, обращайтесь к документации на подключаемый модуль или управляющий элемент ActiveX.

Document.plugins[] — это синоним для Document.embeds[]. Не путайте его с Navigator.plugins[].

fgColor

Строковое свойство, задающее цвет по умолчанию для текста в *document*. Первоначальное значение этого свойства берется из атрибута text тега <body>, и это значение можно установить из сценария в заголовке документа. См. также цветовые свойства объекта HTMLBodyElement в «Справочнике по W3C DOM».

forms[]

Массив объектов Form, по одному для каждой формы HTML, присутствующей в *document*. Массив forms[] имеет forms.length элементов, нумеруемых от нуля до forms.length–1.

images[] *[JavaScript 1.1]*

Массив объектов Image, по одному для каждого изображения, встроенного в документ с помощью тега HTML <img>. Если в теге <img> для изображения задан атрибут name, ссылка на это изображение также сохраняется в свойстве объекта Document. Это свойство имеет то же самое имя, что и изображение. Поэтому, если в изображении имеется атрибут name="toggle", то к изображению можно обращаться с помощью document.toggle.

`lastModified`

Строка, доступная только для чтения, которая указывает дату самого последнего изменения в документе (сообщается веб-сервером). См. справочную страницу «Document.lastModified».

`linkColor`

Строковое свойство, задающее цвет неоткрывавшихся ссылок в документе. Значение этого свойства устанавливается атрибутом `link` тега `<body>`, а также может быть установлено из сценария в заголовке документа. См. также цветовые свойства объекта HTMLBodyElement в «Справочнике по W3C DOM».

`links[]`

Массив объектов Link, по одному на каждую гиперссылку, присутствующую в документе. Массив `links[]` имеет `links.length` элементов, под номерами от нуля до `links.length`–1.

`location` *[признано устаревшим]*

Объект Location, содержащий полный URL текущего документа, синоним свойства `Window.location`. В JavaScript 1.0 это свойство является лишь строковым объектом, доступным только для чтения, которое было предназначено для той же цели, что и свойство `Document.URL`.

`plugins[]` *[JavaScript 1.1]*

Синоним массива `embeds[]`. Ссылается на массив объектов, представляющих плагины или управляющие элементы ActiveX, используемые для отображения встраиваемых данных в документе. Свойство `embeds` является предпочтительным способом доступа к этому массиву, так как оно не создает путаницы с массивом `Navigator.plugins[]`.

`referrer`

Доступная только для чтения строка, содержащая URL документа (если он есть), с которого пользователь пришел к текущему документу. Например, если пользователь открыл в документе A ссылку на документ B, свойство `Document.referrer` в документе B содержит URL документа A. С другой стороны, если пользователь непосредственно ввел URL документа B и не использовал для этого никакую ссылку, свойство `Document.referrer` для документа B равно пустой строке.

`title`

Доступное только для чтения строковое свойство, задающее заголовок текущего документа. Заголовок – это любой текст, расположенный между тегами `<title>` и `</title>` в заголовке документа.

`URL`

Строка, доступная только для чтения, задающая URL документа. См. справочную страницу «Document.URL».

`vlinkColor`

Строковое свойство, задающее цвет открывавшихся ссылок в *`document`*. Значение этого свойства устанавливается атрибутом `vlink` тега `<body>`, и его также можно установить из сценария внутри тега `<head>` документа. См. также цветовые свойства объекта HTMLBodyElement в «Справочнике по W3C DOM».

Свойства в Netscape

`height` *[Netscape 4]*

Высота документа в пикселах.

`layers[]` *[только Netscape 4]*

Массив объектов Layer, представляющих слои, содержащиеся в документе. Каждый объект Layer содержит свой собственный поддокумент, доступный через свойство `document` объекта Layer. Это свойство доступно только в Netscape 4; в Netscape 6 оно не поддерживается.

`width` *[Netscape 4]*

Ширина документа в пикселах.

Свойства в Internet Explorer

`activeElement` *[IE 4]*

Доступное только для чтения свойство, ссылающееся на элемент ввода в документе, который активен в данный момент (то есть имеет фокус ввода).

`all[]` *[IE 4]*

Массив всех элементов в документе. См. справочную страницу «Document.all».

`charset` *[IE 4]*

Наборы символов в документе.

`children[]` *[IE 4]*

Массив, содержащий элементы HTML в порядке их следования в исходном тексте, которые являются непосредственными дочерними элементами документа. Заметьте, что он отличается от массива `all[]`, содержащего все элементы в документе, независимо от их позиций в иерархии вложенности.

`defaultCharset` *[IE 4]*

Набор символов по умолчанию в документе.

`expando` *[IE 4]*

Это свойство, будучи установленным в `false`, не дает возможность расширять клиентские объекты. Другими словами, если программа пытается установить значение несуществующего свойства клиентского объекта, возникает ошибка времени выполнения. Установка `expando` равным `false` иногда помогает находить ошибки, связанные с неверным написанием имен свойств, которые в противном случае трудно обнаружить. Это свойство может быть особенно полезно для программистов, перешедших на JavaScript, но ранее привыкших к языкам, нечувствительным к регистру. Хотя `expando` работает только в IE 4, его можно спокойно (хотя и бесполезно) устанавливать в Netscape.

`parentWindow` *[IE 4]*

Окно, содержащее документ.

`readyState`

Указывает состояние загрузки документа. Оно может иметь одно из следующих четырех строковых значений:

`uninitialized`

Загрузка документа еще не началась.

`loading`

Документ загружается.

`interactive`

Документ загрузился в достаточной степени, чтобы пользователь мог с ним взаимодействовать.

`complete`

Документ полностью загружен.

Методы

Document наследует методы от HTMLElement и определяет следующие методы. Netscape и IE определяют несколько несовместимых методов объекта Document, в основном используемых для DHTML; они перечисляются отдельно.

`clear()`

Удаляет содержимое документа. Этот метод в JavaScript 1.1 признан устаревшим.

`close()`

Закрывает поток документа, открытый с помощью метода `open()`.

`open()`

Открывает поток, в который может записываться содержимое документа.

`write()`

Вставляет указанную строку или строки в анализируемый в данный момент документ или в поток документа, открытый с помощью `open()`.

`writeln()`

Аналогичен `write()`, за исключением того, что дополняет выводимые строки символом перевода строки.

Методы в Netscape

`captureEvents()`

Запрашивает события указанных типов.

`getSelection()`

Возвращает текущий выбранный текст документа.

`releaseEvents`

Прекращает перехват указанных типов событий.

`routeEvent()`

Перенаправляет перехваченное событие к следующему заинтересованному элементу. См. «Window.routeEvent()».

Метод в Internet Explorer

`elementFromPoint()`

Возвращает элемент, расположенный в данной точке (с координатами X и Y).

Обработчик событий

В теге <body> имеются атрибуты onload и onunload. Однако формально обработчики событий onload и onunload принадлежат к объекту Window, а не к объекту Document. См. «Window.onload» и «Window.onunload.»

Синтаксис HTML

Объект Document получает значения для некоторых своих свойств из атрибутов тега HTML <body>. Кроме того, содержимое документа помещается между тегами <body> и </body>:

```
<body
  [ background="imageURL" ]    // Фоновое изображение для документа
  [ bgcolor="цвет" ]           // Фоновый цвет для документа
  [ text="цвет" ]              // Цвет текста для документа
  [ link="цвет" ]              // Цвет для неоткрывавшихся ссылок
  [ alink="цвет" ]             // Цвет для активизируемых ссылок
  [ vlink="цвет" ]             // Цвет для открывавшихся ссылок
  [ onload="обработчик" ]      // JavaScript, запускаемый после загрузки документа
  [ onunload="обработчик" ]    // JavaScript, запускаемый после выгрузки документа
>
// Здесь расположено содержимое документа HTML
</body>
```

Описание

Объект Document представляет документ HTML, отображаемый в окне или фрейме броузера (или в слое в Netscape 4). Свойства этого объекта предоставляют информацию о многих аспектах документа, начиная с цвета текста, фона и якорных элементов, и заканчивая датой последнего изменения. Объект Document также содержит несколько массивов, описывающих содержание документа. Массив links[] содержит по одному объекту Link для каждой гиперссылки в документе. Аналогично, массив applets[] содержит по одному объекту для каждого апплета Java, встроенного в документ, а массив forms[] – по одному объекту Form для каждой формы HTML, присутствующей в документе.

Метод write() объекта Document заслуживает особого упоминания. При вызове в сценариях, запускаемых во время загрузки документа, посредством document.write() можно вставлять динамически сгенерированный текст HTML в документ.

Обзор объекта Document и многих объектов JavaScript, на которые он ссылается, см. в главе 14. Стандарт DOM обсуждается в главе 17.

См. также

Form, свойство document объекта Window; главу 14; Document, HTMLDocument и HTMLBodyElement в «Справочнике по W3C DOM»

Document.all[]

Internet Explorer 4

все элементы HTML в документе

Синтаксис

document.all[*i*]
document.all[*имя*]
document.all.tags(*имя_тега*)

Описание

`all[]` — это универсальный массив, содержащий все элементы HTML в документе. Массив `all[]` содержит элементы в порядке их следования в исходном тексте, их можно извлекать непосредственно из массива, если известно их точное положение в массиве. Однако чаще массив `all[]` применяется для извлечения элементов по значению их атрибутов HTML `name` или `id`. Если указанное имя присвоено более чем одному элементу, то использование этого имени в качестве индекса в массиве `all[]` возвращает массив элементов, разделяющих это имя.

Методу `all.tags()` передается имя тега, и он возвращает массив элементов HTML указанного типа.

См. также

HTMLElement

Document.captureEvents()

См. Window.captureEvents()

Document.clear() JavaScript 1.0; признан устаревшим

очищает документ

Синтаксис

`document``.clear()`

Описание

Метод `clear()` объекта Document признан устаревшим и не должен применяться. Чтобы очистить документ, вы должны просто открыть новый документ с помощью `Document.open()`.

См. также

Document.close(), Document.open(), Document.write()

Document.close() JavaScript 1.0

закрывает выходной поток

Синтаксис

`document``.close()`

Описание

Этот метод отображает все, что было выведено в *`document`*, но еще не отображено, и закрывает выходной поток в *`document`*. Генерируя страницы HTML целиком с помощью `Document.write()`, необходимо по достижении конца страницы вызывать `Document.close()`.

После вызова *`document`*`.close()`, если что-то еще пишется в *`document`* (например, с помощью *`document`*`.write()`), документ неявно очищается и открывается снова, при этом удаляется все, что было записано до вызова метода `close()`.

См. также

Document.open(), Document.write()

Document.cookie

JavaScript 1.0

cookie(s) документа

Синтаксис

document.cookie

Описание

cookie — это строковое свойство, позволяющее читать, создавать, изменять и удалять одно или несколько cookie, применяемых к текущему документу. *cookie* представляет собою небольшой объем именованных данных, сохраняемых веб-броузером. Он дает веб-броузерам «память», и благодаря этому они могут работать с данными, введенными на другой странице, или помнить пользовательские настройки между сессиями веб-броузера. Данные cookie по необходимости автоматически передаются между веб-броузером и веб-сервером, поэтому сценарии CGI на стороне сервера могут читать и писать значения cookie. Клиентский код JavaScript может также читать и писать cookies с помощью этого свойства.

Свойство Document.cookie ведет себя не так, как обычное свойство, доступное для чтения и записи. Значение Document.cookie можно как читать, так и писать, но прочитанное значение, как правило, не совпадает с записанным. Применение этого особенно сложного свойства подробно рассмотрено в главе 16.

Применение

Cookie предназначены для нечастого сохранения небольших объемов данных. Они не подходят в качестве универсального коммуникационного или программного механизма, поэтому используйте их умеренно. Обратите внимание, что веб-броузеры не обязаны сохранять более 20 cookie на один веб-сервер (для всего веб-сервера, а не только для вашего сайта на сервере) и не обязаны сохранять пары cookie *имя/значение* длиной более 4 Кбайт.

См. также

Главу 16

Document.domain

JavaScript 1.1

домен безопасности для документа

Синтаксис

document.domain

Описание

Из соображений безопасности неподписанному сценарию, работающему в одном окне, не разрешается читать свойства из другого окна, если только это окно не использует в качестве хоста тот же веб-сервер. Это создает трудности на больших веб-сайтах, использующих несколько серверов. Например, сценарию на хосте *www.oreilly.com* могут потребоваться свойства, общие со сценарием с хоста *search.oreilly.com*.

С этим помогает справиться свойство domain. Первоначально это строковое свойство содержит имя хоста веб-сервера, с которого был загружен документ, и его можно установить, но только равным доменному суффиксу первоначального значения. Так, сценарий, загруженный с *search.oreilly.com*, может установить собственное свойство

domain равным «oreilly.com». Если в другом окне работает сценарий с *www.oreilly.com* и тоже устанавливает свое свойство domain равным «oreilly.com», эти два сценария могут разделять общие свойства даже несмотря на то, что они получены не с одного сервера.

Заметьте, однако, что сценарий *search.oreilly.com* не может установить свое свойство domain равным «search.oreilly». И что более важно, сценарий с *snoop.spam.com* не может установить свой domain равным «oreilly.com», что могло бы, например, позволить ему определить, какие ключевые слова вы используете.

См. также

Главу 21

Document.elementFromPoint() — Internet Explorer 4

определяет, какой элемент HTML находится в данной точке

Синтаксис

document.elementFromPoint(*x*, *y*)

Аргументы

x – координата X.

y – координата Y.

Возвращаемое значение

Элемент HTML, присутствующий в точке (*x*, *y*) в *document*.

Document.getSelection() — Netscape 4

возвращает выбранный текст

Синтаксис

document.getSelection()

Возвращаемое значение

Текст, если он есть, выделенный в документе в данный момент. Из возвращаемого текста удаляются теги HTML, определяющие форматирование.

Document.handleEvent()

см. Window.handleEvent()

Document.lastModified — JavaScript 1.0

дата изменения документа

Синтаксис

document.lastModified

Описание

lastModified – доступное только для чтения строковое свойство, содержащее дату и время, когда *document* был последний раз изменен. Эти данные получаются из данных

HTTP-заголовка, отсылаемого веб-сервером. Веб-сервер, как правило, получает дату последнего изменения, просматривая дату изменения самого файла.

Веб-серверы не обязаны предоставлять дату последнего изменения для обслуживаемых ими документов. Если веб-сервер не предоставляет дату последнего изменения, JavaScript берет значение 0, которое транслируется в дату, равную полуночи 1 января 1970 года по Гринвичу. Следующий пример показывает, как проверить возникновение такой ситуации.

Пример

Неплохо дать читателям возможность узнать, насколько нова информация, представленная в веб-документе. В документ можно включить автоматическую временную метку, поместив в конце каждого HTML-файла следующий сценарий. Сделав это, вы не должны будете обновлять время изменения вручную при каждом изменении файла. Заметьте, что этот сценарий перед выводом даты проверяет ее на корректность:

```
<script>
if (Date.parse(document.lastModified) != 0)
    document.write('<p><hr><small><i>Последнее изменение: '
                   + document.lastModified
                   + '</i></small>');
</script>
```

См. также

Свойства `location`, `referrer` и `title` объекта Document

Document.links[] JavaScript 1.0

Объекты Link в документе

Синтаксис

document.links
document.links.length

Описание

Свойство `links` — это массив объектов, по одному объекту на каждую гиперссылку, присутствующую в *document*. Массив `links[]` имеет `links.length` элементов, пронумерованных от нуля до `links.length`–1.

См. также

Link

Document.open() JavaScript 1.0

начинает новый документ

Синтаксис

document.open()
document.open(*тип_MIME*)

Аргументы

тип_MIME

Необязательный строковый аргумент, задающий тип данных, записываемых и отображаемых в *document*. Значение этого аргумента должно быть одним из стандартных MIME-типов, понимаемых броузером (для Netscape «text/html», «text/plain», «image/gif», «image/jpeg» и «image/x-bitmap»), или каким-либо другим MIME-типом, которым может управлять установленный плагин. Если этот аргумент опущен, он считается равным «text/html». Этот аргумент игнорируется IE 3, который всегда предполагает для документа тип «text/html». Этот аргумент также не поддерживается в стандартной версии данного метода в W3C DOM. См. статью «HTMLDocument.open()» в «Справочнике по W3C DOM».

Описание

Метод *document*.open() открывает поток в *document*, чтобы последующие вызовы *document*.write() могли дописывать данные в документ. Необязательный аргумент *тип_MIME* задает тип данных, которые должны записываться, и сообщает броузеру о том, как интерпретировать эти данные.

Если любой существующий документ отображается на момент вызова метода open(), при вызове open() или при первом вызове write() или writeln() он автоматически очищается. После открытия документа с помощью open() и записи в него данных с помощью write() документ следует завершить с помощью вызова метода close().

Применение

Обычно Document.open() вызывается без аргументов для открытия документа HTML. Иногда, например, может быть полезен документ типа «text/plain» для использования во всплывающем окне отладочных сообщений.

См. также

Document.close(), Document.write(); HTMLDocument.open() в «Справочнике по W3C DOM»

Document.releaseEvents()

см. Window.releaseEvents()

Document.routeEvent()

см. Window.routeEvent()

Document.URL

JavaScript 1.1

URL текущего документа

Синтаксис

document.URL

Описание

URL — это доступное только для чтения свойство, содержащее полный URL текущего документа.

document.URL обычно равен *window*.location.href окна *window*, содержащего документ *document*. Однако они не всегда равны, поскольку свойство Document.URL может быть изменено путем перенаправления URL, – Window.location содержит запрашиваемый URL, а Document.URL – фактический URL, где был найден документ.

Применение

Некоторые веб-мастера любят включать URL документа в каком-нибудь месте в сам документ, поэтому, например, если документ вставляется в файл или распечатывается, в нем остается ссылка на его местоположение в сети. Следующий сценарий, будучи добавлен в документ, автоматически вставляет URL документа:

```
<script>
document.write('<p><hr><small><i>URL: ' + document.URL
    + '</i></small>');
</script>
```

См. также

Свойства lastModified, location, referrer и title объекта Document; свойство location объекта Window.

Document.write() JavaScript 1.0

дописывает данные в документ

Синтаксис

document.write(*значение*, ...)

Аргументы

значение

Произвольное значение JavaScript, дописываемое к документу *document*. Если значение *value* не является строкой, то оно преобразуется в строку перед дописыванием к документу.

...

Любое количество (ноль и более) дополнительных значений, дописываемых (по порядку) к документу.

Описание

document.write() дописывает каждый из своих аргументов по порядку к *document*. Любые аргументы, не являющиеся строками, преобразуются в строки перед тем, как они будут записаны в конец документа.

Document.write() обычно применяется одним из двух способов. Во-первых, он может вызываться для текущего документа внутри тега <script> или внутри функции, исполняемой при анализе документа. В этом случае метод write() выводит код HTML так, как если он был добавлен в сам файл в том месте, где находится код, вызвавший метод.

Во-вторых, Document.write() позволяет динамически генерировать содержимое документа для окна, отличного от текущего. В этом случае целевой документ не находится в процессе разбора, и поэтому результат не может быть помещен вместо кода, как в вышеописанном случае. Чтобы write() мог вывести текст в документ, этот документ должен быть открыт. Документ можно открыть, явно вызвав метод Document.open().

Однако в большинстве случаев в этом нет необходимости, поскольку при вызове `write()` для закрытого документа этот метод неявно его открывает. При открытии документа любое его содержимое, ранее в нем присутствовавшее, удаляется.

`Document.write()` позволяет дописать любой объем данных в конец открытого документа. Если новый документ целиком генерируется таким способом, он должен быть закрыт посредством метода `Document.close()`. Обратите внимание, что без вызова `open()` обычно можно обойтись, а вызов `close()`обязателен всегда.

Результат вызова `Document.write()` может не сразу отобразиться в целевом окне или фрейме веб-броузера. Дело в том, что веб-броузер может буферизовать данные для вывода большими фрагментами. Вызов `Document.close()` — это единственный способ явно «сбросить» все буферизованные данные и отобразить их в окне броузера.

См. также

Document.close(), Document.open(), Document.writeln(); главу 14

Document.writeln() JavaScript 1.0

дописывает в документ данные и символ перевода строки

Синтаксис

document.writeln(*значение*, ...)

Аргументы

значение

Произвольное значение JavaScript, дописываемое к документу *document*. Если значение *value* не является строкой, оно преобразуется в строку перед дописыванием к документу.

...

Любое количество (ноль и более) дополнительных значений, дописываемых (по порядку) к документу.

Описание

`Document.writeln()` ведет себя точно так же, как `Document.write()`, за исключением того, что после дописывания всех своих аргументов в *document* он также дописывает символ перевода строки. Дополнительную информацию об этом методе можно найти на справочной странице «Document.write()».

Символы перевода строки, как правило, не отображаются в документах HTML, поэтому `Document.writeln()` обычно применяется при записи текста, помещенного в тег `<pre>`, или при записи в документ, открытый с MIME-типом «text/plain».

См. также

Document.close(), Document.open(), Document.write()

Element

см. Input

Event JavaScript 1.2; несовместимые версии поддерживаются в Netscape 4 и IE 4

подробности о событии

Синтаксис

```
function handler(event) { ... }  // Аргумент-обработчик события в Netscape 4
window.event                     // Свойство объекта Window в IE 4
```

Константы

В Netscape 4 объект Event определяет константы битовой маски для каждого из поддерживаемых типов событий. Эти статические свойства используются для образования битовых масок, которые передаются в `captureEvents()` и `releaseEvents()`. Вот доступные константы:

Event.ABORT	Event.BLUR	Event.CHANGE	Event.CLICK
Event.DBLCLICK	Event.DRAGDROP	Event.ERROR	Event.FOCUS
Event.KEYDOWN	Event.KEYPRESS	Event.KEYUP	Event.LOAD
Event.MOUSEDOWN	Event.MOUSEMOVE	Event.MOUSEOUT	Event.MOUSEOVER
Event.MOUSEUP	Event.MOVE	Event.RESET	Event.RESIZE
Event.SELECT	Event.SUBMIT	Event.UNLOAD	

Свойства в Netscape 4

`height`

Установлено только в событиях типа «resize». Задает новую высоту окна или фрейма, размер которого был изменен.

`layerX`, `layerY`

Задает координаты X и Y относительно слоя, где произошло событие.

`modifiers`

Указывает, какие клавиши-модификаторы клавиатуры удерживались, когда произошло событие. Это числовое значение представляет собой битовую маску, состоящую из констант `Event.ALT_MASK`, `Event.CONTROL_MASK`, `Event.META_MASK` или `Event.SHIFT_MASK`. Из-за ошибки это свойство в Netscape 6 и 6.1 не определено.

`pageX`, `pageY`

Задают координаты X и Y относительно страницы веб-броузера, где произошло событие. Заметьте, что это координаты относительно страницы верхнего уровня, а не относительно какого-либо слоя.

`screenX`, `screenY`

Задают координаты X и Y относительно экрана, в котором произошло событие. Заметьте, что в отличие от большинства свойств объекта Event эти свойства поддерживаются и имеют одинаковое значение как в Netscape 4, так и в IE 4.

`target`

Задает объект Window, Document, Layer или HTMLElement, в котором произошло событие.

`type`

Строковое свойство, задающее тип события. Это значение является именем обработчика события минус префикс «on». То есть если вызывается обработчик события `onclick()`, свойство `type` объекта Event будет равно «click».

`which`

Для событий клавиатуры и мыши `which` задает, какая клавиша или кнопка мыши была нажата или отпущена. Для клавиатурных событий это свойство содержит код символа нажатой клавиши. Для событий мыши оно содержит значение 1, 2 или 3, обозначающее левую, среднюю или правую кнопку.

`width`

Установлено только для событий типа «resize». Задает новую ширину окна или фрейма, размер которого был изменен.

`x`, `y`

Задает координаты X и Y места, где произошло событие. В Netscape 4 эти свойства являются синонимами `layerX` и `layerY` и задают позицию относительно содержащего их слоя (если он определен). В IE они имеют другое значение (см. ниже).

Свойства в Internet Explorer 4

`altKey`

Логическое значение, указывающее, удерживалась ли клавиша Alt, когда произошло событие.

`button`

Для событий мыши свойство `button` задает, какая кнопка или кнопки мыши были нажаты. Это доступное только для чтения свойство является битовой маской: бит 1 установлен, если была нажата левая кнопка, бит 2, если была нажата правая кнопка, и бит 4, если была нажата средняя кнопка (трехкнопочной мыши).

`cancelBubble`

Если обработчик события желает остановить распространение события к элементам верхнего уровня, он должен установить это свойство в `true`.

`clientX`, `clientY`

Задает координаты X и Y относительно страницы веб-броузера, где произошло событие.

`ctrlKey`

Логическое значение, указывающее, удерживалась ли клавиша Ctrl при возникновении события.

`fromElement`

Для событий mouseover и mouseout `fromElement` ссылается на объект, из которого был перемещен указатель мыши.

`keyCode`

Для клавиатурных событий `keyCode` задает код символа Unicode, сгенерированного нажатой клавишей.

`offsetX`, `offsetY`

Задает координаты X и Y, где произошло событие, в системе координат исходного элемента события (см. `srcElement`).

`reason`

Для события `datasetcomplete` свойство `reason` содержит код, задающий состояние передачи данных. Значение 0 – успешная передача; 1 – передача была прервана; 2 – во время передачи данных произошла ошибка.

`returnValue`

Если это свойство установлено, его значение имеет приоритет перед значением, фактически возвращаемым обработчиком события. Установите это свойство в `false`, чтобы отменить действие по умолчанию, производимое исходным элементом, в котором произошло событие.

`screenX`, `screenY`

Задает координату X и Y, где произошло событие, относительно экрана. Заметьте, что в отличие от большинства свойств объекта Event эти два свойства поддерживаются и имеют одинаковое значение и в Netscape 4, и в Internet Explorer 4.

`shiftKey`

Логическое значение, указывающее, удерживалась ли клавиша Shift, когда произошло событие.

`srcElement`

Ссылка на объект Window, Document или HTMLElement, сгенерировавший событие.

`srcFilter`

Для событий filterchange `srcFilter` указывает на фильтр, который был изменен.

`toElement`

Для событий mouseover и mouseout `toElement` ссылается на объект, к которому перемещается указатель мыши.

`type`

Строковое свойство, задающее тип события. Его значением является имя обработчика событий минус префикс «on». То есть когда вызывается обработчик события `onclick()`, свойство `type` объекта Event равно «click».

`x`, `y`

Задает координаты X и Y, где произошло событие. В Internet Explorer 4 это свойство задает координату X относительно самого внутреннего содержащего элемента, динамически позиционированного посредством CSS. Netscape 4 интерпретирует эти свойства по-другому (см. предыдущий раздел).

Описание

Объект Event предоставляет подробную информацию о произошедшем событии. К сожалению, эта информация не стандартизована, и объекты Event броузеров Netscape 4 и IE 4 практически полностью несовместимы. Кроме того, что Netscape 4 и IE 4 поддерживают разные свойства, и доступ к объекту в них предоставляется различными способами. В Netscape объект Event передается в качестве аргумента любому обработчику событий. Для обработчиков событий, определяемых с помощью атрибутов HTML, имя аргумента события – `event`. В IE объект Event самого последнего события хранится в свойстве `event` объекта Window.

Мало того, что объекты Event в Netscape 4 и в IE несовместимы, W3C DOM определяет свой собственный объект Event, несовместимый с двумя другими. См. статьи «Event», «UIEvent» и «MouseEvent» в «Справочнике по W3C DOM».

См. также

Главу 19; Event, UIEvent и MouseEvent в «Справочнике по W3C DOM»

FileUpload

JavaScript 1.0

поле загрузки файла для формы **наследуется от Input, HTMLElement**

Синтаксис

form.*имя*
form.elements[*i*]

Свойства

FileUpload наследует свойства от Input и HTMLElement и определяет или перекрывает следующее:

value *[JavaScript 1.1]*

Доступная только для чтения строка, задающая имя файла, введенного пользователем в объект FileUpload. Пользователь может ввести имя файла, либо непосредственно набрав его, либо с помощью броузера каталогов, связанного с объектом FileUpload.

Для того чтобы злонамеренные программы не могли загружать произвольные файлы с клиента, на установку этого свойства из кода JavaScript установлен запрет. По той же причине атрибут value тега <input> не задает начальное значение этого свойства.

Методы

FileUpload наследует методы от Input и HTMLElement.

Обработчик событий

FileUpload наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующие:

onchange

Вызывается, когда пользователь изменяет значение элемента FileUpload и перемещает клавиатурный фокус в другое место. Этот обработчик событий не вызывается для каждого нажатия клавиши в элементе FileUpload, а только когда пользователь завершит редактирование.

Синтаксис HTML

Элемент FileUpload создается с помощью стандартного тега HTML <input>:

```
<form enctype="multipart/form-data"
      method="post">       // Обязательные атрибуты
    ...
  <input
    type="file"            // Указывает, что это элемент FileUpload
    [ name="имя" ]         // Имя, по которому впоследствии можно обращаться к этому элементу
                           // Задает свойство name
```

```
    [ size="целое" ]              // Ширина элемента в символах
    [ maxlength="целое" ]         // Максимально возможное количество вводимых символов
    [ onblur="обработчик" ]       // Обработчик события onblur()
    [ onchange="обработчик" ]     // Обработчик события onchange()
    [ onfocus="обработчик" ]      // Обработчик события onfocus()
  >
    ...
```

Описание

Элемент FileUpload представляет элемент поля ввода для загрузки файла в форме. Это поле ввода во многом похоже на элемент Text. На экране оно выглядит как поле ввода текста, дополненное кнопкой Browse, которая открывает броузер каталогов. Ввод имени файла в элемент FileUpload (непосредственный или выбором в броузере) приводит к тому, что Netscape отсылает содержимое этого файла вместе с формой. Для того чтобы все это работало, в форме должны использоваться кодировка «multipart/form-data» и метод POST.

Элемент FileUpload не имеет свойства `defaultValue` и не признает атрибут HTML `value` для задания начального значения поля ввода. Кроме того, свойство `value` элемента FileUpload доступно только для чтения. Имя может быть введено только пользователем, JavaScript не может каким-либо образом ввести текст в поле FileUpload. Ограничение преследует цель лишить злонамеренные программы JavaScript возможности загружать произвольные файлы (такие как файлы паролей) с пользовательской машины.

См. также

Form, HTMLElement, Input, Text; HTMLInputElement в «Справочнике по W3C DOM»

FileUpload.onchange

JavaScript 1.0

обработчик, вызываемый при изменении введенного значения

Синтаксис

```
<input type="file" onchange="обработчик" ...>
fileupload.onchange
```

Описание

Свойство `onchange` элемента FileUpload задает функцию обработки события, вызываемую, когда пользователь изменяет значение в поле ввода (либо непосредственным набором, либо нажав кнопку Browse) и перемещает фокус ввода в другое место. Этот обработчик предназначен для обработки завершенного изменения введенного значения и, следовательно, не вызывается при каждом нажатии клавиши.

Начальное значение этого свойства – это функция, содержащая инструкции JavaScript, разделенные точками с запятой, которые задаются атрибутом `onchange` определяющего этот объект тега HTML. Когда функция обработки события определяется атрибутом HTML, она исполняется в области действия элемента *fileupload*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onchange`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего элемент *fileupload*.

См. также

Input.onchange; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Form

JavaScript 1.0

форма ввода HTML **наследуется от HTMLElement**

Синтаксис

document.имя_формы
document.forms[*номер_формы*]

Свойства

Наследует свойства от HTMLElement и определяет или перекрывает следующие:

`action`

Строка, доступная для чтения и записи (в IE 3 – только для чтения), задающая URL, по которому отсылаются данные формы. Начальное значение этого свойства указывается атрибутом `action` тега HTML `<form>`. Обычно этот URL задает адрес сценария CGI, хотя может также быть адресом `mailto:` или `news:`.

`elements[]`

Массив элементов ввода, присутствующих в форме. Каждый элемент – это объект Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text или Textarea. См. справочную страницу «Form.elements».

`encoding`

Строка, доступная для чтения и записи (в IE 3 – только для чтения), задающая способ кодирования данных формы при передаче данных из нее. Начальное значение этого свойства задается атрибутом `enctype` тега `<form>`. По умолчанию имеет значение «application/x-www-form-urlencoded», которого достаточно почти для всех применений. Но есть и другие значения. Например, «text/plain» удобно, когда форма отправляется по email на URL `mailto:`. Подробно этот вопрос рассматривается в книге «CGI Programming on the World Wide Web» Шишира Гундаварама (Shishir Gundavaram), O'Reilly.[1]

`length`

Количество элементов формы. Эквивалентно `elements.length`.

`method`

Строка, доступная для чтения и записи (в IE 3 – только для чтения), задающая метод, с помощью которого выполняется передача данных формы. Начальное значение этого свойства задается атрибутом `method` тега `<form>`. Допустимые значения – `get` и `post`.

`get` – это метод, используемый по умолчанию. Обычно он применяется для таких видов передачи данных, как запросы к базе данных, которые не имеют побочных эффектов. В этом методе закодированные данные формы дописываются к URL, заданному свойством `Form.action`. Метод `post` подходит для записи данных в БД,

[1] Гулич С., Гундаварам Ш., Бирзнекс Г. «CGI программирование на Perl», 2-е издание. – Пер. с англ. – СПб: Символ-Плюс, 2001.

имеющей побочные эффекты. При этом закодированные данные отсылаются в теле HTTP-запроса.

name

Задает имя формы. Начальное значение этого доступного для чтения и записи строкового свойства берется из атрибута name тега <form>.

target

Строка, доступная для чтения и записи, задающая имя фрейма или окна, в котором должны быть показаны результаты передачи данных формы. Начальное значение определяется атрибутом target. Для свойства target и атрибута target также поддерживаются специальные имена "_top", "_parent", "_self" и "_blank". См. справочную страницу «Form.target».

Методы

Форма наследует методы от HTMLElement и определяет следующие методы:

reset()

Сбрасывает все элементы ввода формы к их значениям по умолчанию.

submit()

Передает данные из формы.

Обработчик событий

Форма наследует обработчики событий от HTMLElement и определяет следующие методы:

onreset

Вызывается непосредственно перед сбросом элементов формы. В HTML задается атрибутом onreset.

onsubmit

Вызывается непосредственно перед передачей данных формы. В HTML задается атрибутом onsubmit. Этот обработчик события позволяет выполнить проверку значений формы перед их передачей на сервер.

Синтаксис HTML

Объект Form создается с помощью стандартного тега HTML <form>. Форма содержит любые элементы ввода, создаваемые тегами <input>, <select> и <textarea> между <form> и </form>:

```
<form
    [ name="имя_формы" ]          // Используется в JavaScript для присваивания имени форме
    [ target="имя_окна" ]         // Имя окна для откликов
    [ action="url" ]              // URL, куда отправляются данные формы
    [ method=("get"|"post") ]     // Метод передачи данных формы
    [ enctype="кодировка" ]       // Как кодируются данные формы
    [ onreset="обработчик" ]      // Обработчик, вызываемый при сбросе формы
    [ onsubmit="обработчик" ]     // Обработчик, вызываемый при передаче данных формы
>
// Здесь находятся текст и элементы ввода формы
</form>
```

Описание

Объект Form представляет форму HTML в документе. Каждая форма в документе представлена элементом массива `Document.forms[]`. Именованные формы также представлены свойством *имя_формы* соответствующего объекта Document, где *имя_формы* – это имя, заданное в атрибуте `name` тега `<form>`.

Элементы формы (кнопки, поля ввода, переключатели и т. д.) собраны в массиве `Form.elements[]`. К именованным элементам, как и к именованным формам, можно обращаться непосредственно по имени – имя элемента выступает в качестве имени свойства объекта Form. Другими словами, обратиться к элементу объекта Text с именем `phone` в форме с именем `questionnaire` можно посредством следующего выражения JavaScript:

```
document.questionnaire.phone
```

См. также

Button, Checkbox, FileUpload, Hidden, Input, Password, Radio, Reset, Select, Submit, Text, Textarea; главу 15; HTMLFormElement в «Справочнике по W3C DOM»

Form.elements[] JavaScript 1.0

элементы ввода формы

Синтаксис

form`.elements[`*i*`]`
form`.elements.length`

Описание

form`.elements[]` – это массив объектов ввода в форме *form*. Массив имеет `elements.length` элементов. Тип элемента может быть любым, соответствующим элементам ввода: Button, Checkbox, Hidden, Password, Radio, Reset, Select, Submit, Text и Textarea. Эти объекты ввода расположены в массиве в том же порядке, что и в исходном коде формы HTML.

Применение

Если элементу массива *form*`.elements[]` было дано имя с помощью атрибута `name="`*имя*`"` его тега HTML `<input>`, это имя становится свойством объекта *form*, и это свойство ссылается на элемент. То есть можно ссылаться на объекты ввода по имени, а не по номеру:

```
form.имя
```

Обращаться к элементам по имени обычно проще, поэтому предпочтительно указывать атрибут `name` для всех элементов формы.

См. также

Button, Checkbox, Form, Hidden, Input, Password, Radio, Reset, Select, Submit, Text, Textarea

Form.onreset

JavaScript 1.1

обработчик, вызываемый при сбросе формы

Синтаксис

```
<form ... onreset="обработчик" ... >
form.onreset
```

Описание

Свойство onreset объекта Form задает функцию обработки события, вызываемую, когда пользователь щелкает по кнопке Reset формы или когда вызывается метод Form.reset().

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, разделенные точками с запятой, которые задаются атрибутом onreset тега HTML <form>. Если функция-обработчик определяется атрибутом HTML, она выполняется в области действия элемента *form*, а не в области действия содержащего его окна.

В модели событий Netscape 4 объект Event передается в качестве аргумента функции-обработчику onreset. В модели событий IE аргумент не передается, но соответствующий объект Event доступен как свойство event объекта Window, содержащего элемент *form*.

Если обработчик onreset возвращает false, элементы формы не сбрасываются.

Пример

Следующий обработчик событий запрашивает пользователя, действительно ли он хочет очистить форму:

```
<form ...
    onreset="return confirm('Вы действительно хотите удалить все введенные данные?')"
>
```

См. также

Form.onsubmit, Form.reset(); главу 15; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Form.onsubmit

JavaScript 1.0

вызывается при передаче данных формы

Синтаксис

```
<form ... onsubmit="обработчик" ... >
form.onsubmit
```

Описание

Свойство onsubmit объекта Form задает функцию обработки события, вызываемую, когда пользователь передает данные из формы на сервер щелчком по кнопке Submit формы. Обратите внимание, что этот обработчик события не вызывается при вызове метода Form.submit().

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, разделенные точками с запятой, которые задаются атрибутом onsubmit тега HTML <form>. Если функция обработки события определяется атрибу-

том HTML, то она исполняется в области действия элемента *form*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onsubmit. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего элемент *form*.

Если обработчик события onsubmit возвращает false, элементы формы не сохраняются. Если обработчик возвращает любое другое значение или не возвращает ничего, форма передается обычным образом. Так как обработчик onsubmit может отменить передачу данных формы, он идеален для выполнения проверки данных формы.

См. также

Form.onreset, Form.submit(); главу 15; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Form.reset() JavaScript 1.1

сбрасывает элементы формы

Синтаксис

form.reset()

Описание

Метод reset() сбрасывает указанную форму, восстанавливая все ее элементы к значениям по умолчанию, в точности так же, как если бы пользователь щелкнул по кнопке Reset. Сначала вызывается обработчик события onreset() формы, который может предотвратить сброс формы, вернув значение false.

Form.submit() JavaScript 1.0

передача данных формы

Синтаксис

form.submit()

Описание

Метод submit() передает данные из указанной формы *form* почти так же, как если бы пользователь щелкнул по кнопке Submit. Данные формы передаются так, как это указано в свойствах action, method и encoding элемента *form* (или в атрибутах action, method и enctype тега <form>), а результаты отображаются в окне или фрейме, заданном свойством target или атрибутом target.

Важное различие между вызовом метода submit() и передачей данных формы состоит в том, что при вызове submit() обработчик события onsubmit() не вызывается. Например, если onsubmit() применяется для проверки введенных данных, то проверку необходимо выполнять явно перед вызовом submit().

Применение

Чаще применяется не непосредственный вызов метода submit(), а кнопка Submit, нажав которую, пользователь может подтвердить форму.

Form.target

JavaScript 1.0; в Internet Explorer 3 доступно только для чтения

окно для результатов формы

Синтаксис

form.target

Описание

target — это доступное для чтения и записи строковое свойство объекта Form. Оно задает имя фрейма или окна, в котором должны отображаться результаты передачи данных формы. Начальное значение этого свойства задается атрибутом target тега <form>. Если он не установлен, то по умолчанию результаты передачи данных формы выводятся в том же окне, в котором находится форма.

Заметьте, что значением target является имя фрейма или окна, а не сам фрейм или окно. Имя фрейма задается атрибутом name тега <frame>. Имя окна указывается при создании окна с помощью вызова метода Window.open(). Если target задает окно, которое не существует, броузер автоматически создает новое окно для отображения результатов передачи данных формы, и любые последующие формы с тем же именем атрибуте target используют то же самое только что созданное окно.

Для target поддерживается четыре специальных имени. Значение "_blank" указывает, что должно быть открыто новое пустое окно броузера, предназначенное для результатов передачи данных формы. Значение "_self" принято по умолчанию; оно указывает, что результаты передачи данных формы должны отображаться в том же фрейме или окне, что и сама форма. Значение "_parent" указывает, что результаты должны отображаться в родительском фрейме фрейма, содержащего форму. И наконец, значение "_top" указывает, что результат должен отображаться во фрейме самого верхнего уровня — другими словами, все фреймы должны быть удалены и результаты должны занимать окно броузера целиком.

Это свойство можно установить в IE 3, но это не влияет на фактическое целевое окно или фрейм формы.

См. также

Link.target

Frame

JavaScript 1.0

тип объекта Window

Синтаксис

window.frames[*i*]
window.frames.length
frames[*i*]
frames.length

Описание

Несмотря на то что иногда упоминается объект Frame, строго говоря, такого объекта не существует. Все фреймы внутри окна броузера представляют собой экземпляры объекта Window, и они содержат те же свойства и поддерживают те же методы и обработчики событий, что и объект Window. Подробности см. в статьях по объекту Window, его свойствам, методам и обработчикам событий.

Между объектами Window, представляющими окна броузера верхнего уровня, и теми, что представляют фреймы в окне броузера, имеется несколько практических различий:

- Когда свойство `defaultStatus` для фрейма установлено, указанное сообщение в строке состояния видимо, только когда курсор мыши находится внутри фрейма.
- Свойства `top` и `parent` окна верхнего уровня всегда ссылаются на само окно. Эти свойства реально полезны только для фреймов.
- Метод `close()` не представляет пользы для объектов Window, являющихся фреймами.

См. также

Window

getClass()

Netscape 3 LiveConnect

возвращает JavaClass для JavaObject

Синтаксис

`getClass(`*`javaobj`*`)`

Аргументы

`javaobj`

Объект JavaObject.

Возвращаемое значение

Объект JavaClass для *`javaobj`*.

Описание

`getClass()` – это функция, принимающая в качестве аргумента объект JavaObject (*`javaobj`*). Она возвращает объект JavaClass данного JavaObject. То есть она возвращает объект JavaClass, представляющий Java-класс объекта Java, представленного указанным JavaObject.

Применение

Не путайте функцию `getClass()` JavaScript с методом *getClass* всех объектов Java. Аналогично, не следует путать объект JavaClass в JavaScript с классами *java.lang.Class* в Java.

Рассмотрим объект Java Rectangle, созданный с помощью следующей строки:

```
var r = new java.awt.Rectangle();
```

`r` – это переменная JavaScript, в которой хранится объект JavaObject. Вызов функции `getClass()` JavaScript возвращает объект JavaClass, представляющий класс *java.awt.Rectangle*:

```
var c = getClass(r);
```

Это можно увидеть, сравнив данный объект JavaClass с `java.awt.Rectangle`:

```
if (c == java.awt.Rectangle) ...
```

Метод `getClass()` в Java вызывается по-другому и выполняет совершенно другую функцию:

```
c = r.getClass();
```

После исполнения приведенной выше строки кода c есть не что иное, как JavaObject, который представляет объект java.lang.Class. Это объект Java, являющийся Java-представлением класса *java.awt.Rectangle*. Подробную информацию о том, что можно делать с классом *java.lang.Class*, см. в документации по Java.

Итак, мы видим, что следующее выражение всегда равно true для любого JavaObject o:

```
(getClass(o.getClass()) == java.lang.Class)
```

См. также

JavaArray, JavaClass, JavaObject, JavaPackage, свойство java объекта Window; главу 22

Hidden

JavaScript 1.0; расширен в JavaScript 1.1

скрытые данные для взаимодействия между клиентом и сервером — **наследуется от Input, HTMLElement**

Синтаксис

form.имя
form.elements[*i*]

Свойства

Hidden наследует свойства от Input и HTMLElement и определяет или перекрывает следующие свойства:

value

Строка, доступная для чтения и записи, которая задает произвольные данные, передаваемые на веб-сервер при передаче данных формы, содержащей объект Hidden. Начальное значение задается атрибутом value тега <input>, который определяет объект Hidden.

Синтаксис HTML

Элемент Hidden создается с помощью стандартного тега HTML <input>:

```
<form>
    ...
  <input
    type="hidden"        // Указывает, что это элемент Hidden
    [ name="имя" ]       // Имя, по которому впоследствии можно обращаться к этому элементу
                         // Задает значение свойства name
    [ value="значение" ] // Значение, передаваемое при отправке данных формы
                         // Задает начальное значение свойства value
  >
    ...
</form>
```

Описание

Элемент Hidden – это невидимый, скрытый элемент формы, позволяющий передавать произвольные данные на сервер при передаче данных формы. Посредством элемента Hidden можно передать на сервер информацию, отличную от данных, введенных пользователем.

Когда HTML-документ создается сервером «на лету», еще одним применением скрытых элементов форм является передача данных с сервера клиенту для последующей обработки JavaScript на стороне пользователя. Например, сервер может передать клиенту исходные данные в компактном внутреннем формате, указав данные в атрибуте `value` одного или нескольких элементов Hidden. На стороне клиента программа JavaScript (переданная вместе с данными или в другом фрейме) может прочитать свойство `value` одного или нескольких элементов Hidden, обработать, отформатировать и вывести эти данные в менее компактном, понятном человеку (и, возможно, настраиваемом им) формате.

Скрытые элементы могут также быть полезны для взаимодействия между сценариями CGI, даже без вмешательства JavaScript на стороне клиента. При таком применении один CGI-сценарий генерирует динамическую страницу HTML, содержащую скрытые данные, которые затем отсылаются другому сценарию CGI. Эти скрытые данные могут передавать информацию о состоянии, такую как результаты передачи данных предыдущей формы.

Передача информации от клиента на сервер также может быть организована при помощи cookie. Однако важное различие между скрытыми элементами форм и cookies состоит в том, что cookies сохраняются на стороне клиента.

См. также

Document.cookie, Form, HTMLElement, Input; HTMLInputElement в «Справочнике по W3C DOM»

History

JavaScript 1.0

URL-история броузера

Синтаксис

window.history
frame.history
history

Свойства

length

Это числовое свойство задает количество URL в списке истории броузера. Знание размера этого списка не особенно полезно, поскольку нет способа определить индекс текущего отображаемого документа в этом списке.

Методы

back()

Переходит назад к ранее открывавшемуся URL.

forward()

Переходит вперед к ранее открывавшемуся URL.

go()

Переходит к ранее открывавшемуся URL.

Описание

Объект History представляет историю просмотра окна – он поддерживает список ранее открывавшихся веб-страниц. Однако по соображениям безопасности и секретности содержимое этого списка недоступно для сценариев. Несмотря на то что сценарии не могут обращаться к URL, хранящимся в объекте History, они могут использовать свойство `length` для определения числа URL в списке и методы `back()`, `forward()` и `go()` для того, чтобы броузер мог заново открыть любой URL из массива.

Пример

Следующая строка выполняет то же действие, что и щелчок по кнопке Back броузера:

```
history.back();
```

Следующий код выполяет то же действие, что и два щелчка по кнопке Back:

```
history.go(-2);
```

См. также

Свойство `history` объекта Window, Location

History.back() JavaScript 1.0

возврат к предыдущему URL

Синтаксис

`history.back()`

Описание

В результате вызова `back()` окно или фрейм, которому принадлежит объект History, заново открывает URL (если он есть), который был открыт непосредственно перед текущим. Вызов этого метода имеет тот же эффект, что и щелчок по кнопке Back в броузере. Он также эквивалентен:

```
history.go(-1);
```

History.forward() JavaScript 1.0

открывает следующий URL

Синтаксис

`history.forward()`

Описание

В результате вызова `forward()` окно или фрейм, которому принадлежит объект History, заново открывает URL (если он есть), который был открыт непосредственно после текущего. Вызов этого метода имеет тот же эффект, что и щелчок по кнопке Forward в броузере. Он эквивалентен:

```
history.go(1);
```

Обратите внимание, что если пользователь не перемещался назад по истории просмотра при помощи кнопки Back или меню Go и если JavaScript не вызывал методы `History.back()` или `History.go()`, то метод `forward()` не действует, так как броузер уже находится в конце своего списка URL и нет адреса, к которому можно было бы перейти.

History.go()

JavaScript 1.0; расширен в JavaScript 1.1

повторно открывает URL

Синтаксис

history.go(*относительная_позиция*)
history.go(*целевая_строка*)

Аргументы

относительная_позиция

Относительная позиция в списке историй URL, которую следует открыть. В IE 3 этот аргумент должен быть равен 1, 0 или –1.

целевая_строка

Подстрока URL, который должен быть открыт. Эта версия метода go() была добавлена в JavaScript 1.1.

Описание

Первая форма метода History.go() принимает целый аргумент и заставляет броузер открыть URL, который находится на указанном расстоянии в списке истории, хранящемся в объекте History. Положительные аргументы перемещают броузер вперед по списку, а отрицательные – назад. Другими словами, вызов history.go(-1) эквивалентен вызову history.back() и действует так же, как щелчок по кнопке Back. Аналогично, history.go(3) заново открывает тот же URL, который был бы открыт при трехкратном вызове history.forward(). Вызов go() с аргументом, равным 0, приводит к повторной загрузке текущей страницы (хотя в Netscape 3 Location.reload() предоставляет лучший способ сделать это). В многофреймовых документах в Netscape 3 эта форма метода содержит ошибку, а в IE он может быть вызван только со значениями 1, 0 и –1.

Вторая форма метода History.go() была реализована в JavaScript 1.1. Она принимает строковый аргумент и приводит к тому, что броузер повторно открывает первый (то есть позднее всего открытый) URL, содержащий указанную строку.

HTMLElement

JavaScript 1.2

суперкласс всех элементов HTML

Синтаксис

HTMLElement – это суперкласс всех классов, представляющих элементы HTML. Поэтому объекты HTMLElement выступают в клиентском JavaScript во многих контекстах и доступны любым из следующих способов:

document.images[*i*]
document.links[*i*]
document.anchors[*i*]
document.forms[*i*]
document.forms[*i*].elements[*j*]
document.*имя_элемента*
document.*имя_формы*.*имя_элемента*
document.all[*i*]

Свойства

`all[]` *[IE 4]*

Полный список всех элементов в исходном порядке, содержащихся в этом элементе. Это свойство ведет себя точно так же, как свойство `Document.all[]`. См. справочную страницу «Document.all[]».

`children[]` *[IE 4]*

Элементы, являющиеся непосредственными дочерними элементами данного элемента.

`className` *[IE 4, Netscape 6]*

Строка, доступная для чтения и для записи, задающая значение атрибута `class` элемента. Это свойство применяется совместно с каскадными таблицами стилей.

`document` *[IE 4]*

Ссылка на содержащий этот элемент объект Document.

`id` *[IE 4, Netscape 6]*

Доступная для чтения и записи строка, задающая значение атрибута `id` данного элемента. Это свойство используется для присваивания элементу уникального имени.

`innerHTML` *[IE 4, Netscape 6]*

Строка, доступная для чтения и для записи, задающая текст HTML, который содержится внутри элемента, исключая открывающий и закрывающий теги самого элемента. Установка этого свойства заменяет содержимое элемента указанным текстом HTML. Заметьте, что это свойство нельзя установить, пока документ загружается.

`innerText` *[IE 4]*

Строка, доступная для чтения и записи, задающая простой текст, который содержится внутри элемента, исключая открывающий и закрывающий теги самого элемента. Установка свойства заменяет содержимое элемента неанализируемым обычным текстом. Заметьте, что пока документ загружается, установить это свойство нельзя.

`lang` *[IE 4, Netscape 6]*

Строка, доступная для чтения и записи, задающая значение атрибута HTML `lang` элемента.

`offsetHeight` *[IE 4]*

Высота элемента и его содержимого в пикселах.

`offsetLeft` *[IE 4]*

Координата X элемента относительно элемента-контейнера `offsetParent`.

`offsetParent` *[IE 4]*

Задает элемент-контейнер, определяющий систему координат, в которой измеряются `offsetLeft` и `offsetTop`. Для большинства элементов `offsetParent` – это объект Document, который их содержит. Однако если у элемента есть динамически позиционированный контейнер, то этим элементом является `offsetPartner`. Аналогично, клетки таблицы позиционируются относительно строки, в которой они содержатся.

`offsetTop` *[IE 4]*

Координата Y элемента относительно элемента-контейнера `offsetParent`.

`offsetWidth` *[IE 4]*

Ширина элемента и всего его содержимого в пикселах.

`outerHTML` *[IE 4]*

Свойство, доступное для чтения и записи, задающее текст HTML для элемента, включая его начальный и конечный теги. Установка этого свойства в строке текста HTML полностью заменяет элемент и его содержимое. Заметьте, что нельзя установить это свойство, пока документ загружается.

`outerText` *[IE 4]*

Свойство, доступное для чтения и для записи, задающее текст элемента, включая его начальный и конечный теги. Установка этого свойства полностью заменяет элемент и его содержимое указанным простым текстом. Заметьте, что установить это свойство, пока документ загружается, невозможно.

`parentElement` *[IE 4]*

Элемент, являющийся непосредственным родителем данного элемента. Это свойство доступно только для чтения.

`sourceIndex` *[IE 4]*

Индекс элемента в массиве `Document.all[]` документа, в котором он содержится.

`style` *[IE 4, Netscape 6]*

Встроенные стилевые атрибуты CSS этого элемента. Установка свойств этого объекта изменяет стиль отображения элемента. См. главу 18.

`tagName` *[IE 4, Netscape 6]*

Доступная только для чтения строка, задающая имя тега HTML, которым определен элемент.

`title` *[IE 4, Netscape 6]*

Строка, доступная для чтения и записи, задающая значение атрибута `title` тега HTML, которым определен элемент. Большинство броузеров используют эту строку как «всплывающую подсказку» для элемента.

Методы

`contains()`	Определяет, содержится ли в данном элементе указанный элемент.
`getAttribute()`	Получает значение атрибута с указанным именем.
`handleEvent()`	Передает объект Event соответствующему обработчику события.
`insertAdjacentHTML()`	Вставляет текст HTML рядом с данным элементом.
`insertAdjacentText()`	Вставляет простой текст в документ около данного элемента.
`removeAttribute()`	Удаляет атрибут и его значение из элемента.
`scrollIntoView()`	Прокручивает документ, чтобы элемент был видим наверху или внизу окна.
`setAttribute()`	Устанавливает значение атрибута элемента.

Обработчики событий

`onclick`	Вызывается, когда пользователь щелкает по элементу.
`ondblclick`	Вызывается, когда пользователь дважды щелкает по элементу.
`onhelp`	Вызывается, когда пользователь запрашивает помощь. Только в IE 4.
`onkeydown`	Вызывается, когда пользователь нажимает клавишу.
`onkeypress`	Вызывается, когда пользователь нажимает и отпускает клавишу.
`onkeyup`	Вызывается, когда пользователь отпускает клавишу.
`onmousedown`	Вызывается, когда пользователь нажимает кнопку мыши.
`onmousemove`	Вызывается, когда пользователь перемещает мышь.
`onmouseout`	Вызывается, когда пользователь уводит курсор мыши из элемента.
`onmouseover`	Вызывается, когда пользователь перемещает курсор мыши над элементом.
`onmouseup`	Вызывается, когда пользователь отпускает кнопку мыши.

Описание

HTMLElement – это суперкласс для всех классов JavaScript, представляющих элементы HTML: Anchor, Form, Image, Input, Link и т. д. HTMLElement определяет обработчики событий, которые реализованы во всех элементах как в IE 4, так и в Netscape 4. Объектная модель документа IE 4 предоставляет доступ ко всем элементам HTML в документе, поэтому она определяет довольно много свойств и методов для этих элементов. Netscape 4 не реализует ни одно из этих свойств и методов IE (за исключением `handleEvent()`, специфического для Netscape), но Netscape 6 реализует те из них, которые были стандартизованы в W3C DOM. Полную информацию по стандартным свойствам и методам элементов HTML см. в «Справочнике по W3C DOM».

См. также

Anchor, Form, Image, Input, Link; главу 17; главу 19; Element, HTMLElement и Node в «Справочнике по W3C DOM»

HTMLElement.contains()

Internet Explorer 4

содержится ли один элемент в другом элементе

Синтаксис

элемент`.contains(`*цель*`)`

Аргументы

цель – объект HTMLElement.

Возвращаемое значение

`true`, если *элемент* содержит *цель*; `false`, если не содержит.

HTMLElement.getAttribute()

Internet Explorer 4, Netscape 6

получение значения атрибута

Синтаксис

элемент`.getAttribute(`*имя*`)`

Аргументы

имя – имя атрибута.

Возвращаемое значение

Значение именованного атрибута *элемента* или `null`, если *элемент* не имеет атрибута с указанным *именем*.

HTMLElement.handleEvent()

см. Window.handleEvent()

HTMLElement.insertAdjacentHTML() Internet Explorer 4

вставляет текст HTML рядом с элементом

Синтаксис

элемент.`insertAdjacentHTML(`*где*, *текст*`)`

Аргументы

где

Строка, указывающая точку вставки текста. Если она принимает значение «BeforeBegin», то *текст* должен быть вставлен непосредственно перед начальным тегом *элемента*. Если «AfterBegin», то *текст* должен быть вставлен непосредственно после начального тега *элемента*. «BeforeEnd» означает, что *текст* должен быть вставлен непосредственно перед конечным тегом *элемента*, а «AfterEnd» – *текст* должен быть вставлен непосредственно после конечного тега *элемента*.

текст

Вставляемый HTML.

Описание

`insertAdjacentHTML()` вставляет код HTML *текст* в позиции внутри или рядом с *элементом*, как указано аргументом *где*.

HTMLElement.insertAdjacentText() Internet Explorer 4

вставляет обычный текст перед или после элемента

Синтаксис

элемент.`insertAdjacentText(`*где*, *текст*`)`

Аргументы

где

Строка, указывающая, куда вставляется текст. Если она принимает значение «BeforeBegin» то *текст* должен быть вставлен непосредственно перед начальным тегом *элемента*. «AfterBegin» означает, что *текст* должен быть вставлен непосредственно после начального тега *элемента*, а «BeforeEnd» – *текст* должен быть вставлен непосредственно перед конечным тегом *элемента*. Если же строка равна «AfterEnd», то *текст* должен быть вставлен непосредственно после конечного тега *элемента*.

текст

Вставляемый обычный текст.

Описание

insertAdjacentText() вставляет обычный текст *текст* в позиции внутри или рядом с *элементом*, как указано аргументом *где*.

HTMLElement.onclick

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при щелчке по элементу

Синтаксис

```
<элемент onclick="обработчик" ... >
элемент.onclick
```

Описание

Свойство onclick объекта HTMLElement, задающее функцию обработки события, вызываемую, когда пользователь щелкает по *элементу*. Обратите внимание, что onclick отличается от onmousedown. Событие click никогда не происходит, если событие mousedown и последующее событие mouseup не произошли над одним и тем же *элементом*.

Начальное значение этого свойства – это функция, содержащая инструкции JavaScript, которые задаются атрибутом onclick определяющего объект тега HTML. Когда функция обработки события определяется атрибутом HTML, она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onclick. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего *элемент*.

В Netscape 4 свойство Event.which задает, какая кнопка мыши была нажата. В IE 4 номер кнопки задается свойством Event.button.

См. также

Event, Input.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.ondblclick

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при двойном щелчке по элементу

Синтаксис

```
<элемент ondblclick="обработчик" ... >
элемент.ondblclick
```

Описание

Свойство ondblclick объекта HTMLElement задает функцию обработки события, которая вызывается, когда пользователь делает двойной щелчок по *элементу*.

Начальное значение этого свойства – это функция, содержащая инструкции JavaScript, которые задаются атрибутом ondblclick определяющего объект тега HTML. Когда функция обработки события определяется атрибутом HTML, она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику ondblclick. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего *элемент*.

См. также

Event; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.onhelp

Internet Explorer 4

обработчик, вызываемый при нажатии клавиши F1

Синтаксис

```
<элемент onhelp="обработчик" ... >
элемент.onhelp
```

Описание

Свойство onhelp *элемента* задает функцию-обработчик событий, которая вызывается при нажатии пользователем клавиши F1, когда *элемент* имеет клавиатурный фокус.

Начальное значение этого свойства – это функция, содержащая инструкции JavaScript, которые задаются атрибутом onhelp определяющего объект тега HTML. Когда функция обработки события определяется атрибутом HTML, она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

После вызова функции обработчика onhelp Internet Explorer 4 отображает встроенное окно помощи.

HTMLElement.onkeydown

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при нажатии клавиши

Синтаксис

```
<элемент onkeydown="обработчик" ... >
элемент.onkeydown
```

Описание

Свойство onkeydown объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь нажимает клавишу над *элементом*.

Начальное значение этого свойства – это функция, содержащая инструкции JavaScript, которые задаются атрибутом ondblkeydown определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onkeydown. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего *элемент*.

Код символа нажатой клавиши содержится в свойстве which объекта Event в Netscape и в свойстве keyCode объекта Event в IE. Этот код клавиши можно преобразовать в строку с помощью метода String.fromCharCode(). Действующие в данный момент кла-

виши-модификаторы могут быть определены из свойства Event.modifiers в Netscape или с помощью метода Event.shiftKey() и сходных с ним методов в IE.

В событийной модели Netscape можно отменить обработку нажатия клавиши, вернув из этого обработчика false. В событийной модели IE можно отменить обработку, установив Event.returnValue в false. В IE этот обработчик может вернуть альтернативный код клавиши, который будет использоваться вместо кода фактически нажатой клавиши.

Часто вместо обработчиков onkeydown и onkeyup применяется обработчик onkeypress.

См. также

Event, HTMLElement.onkeypress; главу 19

HTMLElement.onkeypress

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при нажатии клавиши

Синтаксис

```
<элемент onkeypress="обработчик" ... >
элемент.onkeypress
```

Описание

Свойство onkeypress объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь нажимает клавишу над *элементом*. Событие keypress генерируется после события keydown, но перед соответствующим событием keyup. События keypress и keydown схожи между собой. Если нет необходимости в получении индивидуальных событий keyup, то следует предпочесть свойство onkeypress (а не onkeydown).

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, которые задаются атрибутом onkeypress определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onkeypress. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего *элемент*.

Код символа нажатой клавиши содержится в свойстве which объекта Event в Netscape и в свойстве keyCode объекта Event в IE. Этот код клавиши можно преобразовать в строку с помощью метода String.fromCharCode(). Действующие в данный момент клавиши-модификаторы могут быть определены из свойства Event.modifiers в Netscape или с помощью метода Event.shiftKey() и сходных ему методов в IE.

В событийной модели Netscape можно отменить обработку нажатия клавиши, вернув из этого обработчика false. В событийной модели IE обработку можно отменить, установив Event.returnValue в false. В IE этот обработчик может вернуть альтернативный код клавиши, который будет взят вместо кода фактически нажатой пользователем клавиши.

См. также

Event; главу 19

HTMLElement.onkeyup

JavaScript 1.2; HTML 4.0

обработчик, вызываемый, когда пользователь отпускает клавишу

Синтаксис

```
<элемент onkeyup="обработчик" ... >
элемент.onkeyup
```

Описание

Свойство `onkeyup` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь отпускает клавишу над *элементом*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onkeyup` определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onkeyup`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *элемент*.

См. также

Event, HTMLElement.onkeydown; главу 19

HTMLElement.onmousedown

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при нажатии кнопки мыши

Синтаксис

```
<элемент onmousedown="обработчик" ... >
элемент.onmousedown
```

Описание

Свойство `onmousedown` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь нажимает кнопку мыши над *элементом*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onmousedown` определяющего объект тега HTML. Когда функция обработки события определяется атрибутом HTML, она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onmousedown`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *элемент*.

В Netscape свойство `Event.which` определяет, какая кнопка мыши была нажата. В IE номер кнопки мыши определяется с помощью свойства `Event.button`.

См. также

Event, HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.onmousemove

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при перемещении курсора мыши внутрь элемента

Синтаксис

```
<элемент onmousemove="обработчик" ... >
элемент.onmousemove
```

Описание

Свойство `onmousemove` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользовтель перемещает курсор мыши внутрь *элемента*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onmousemove` определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onmousemove`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *элемент*.

Если определить обработчик события `onmousemove`, то события перемещения мыши генерируются в огромных количествах при перемещении мыши внутри *элемента*. Имейте это в виду при написании функции, которая должна вызываться этим обработчиком.

В Netscape 4 этот обработчик события нельзя определить для отдельных элементов, вместо этого необходимо явно зарегистрировать заинтересованность в событиях перемещения мыши, перехватив их с помощью метода `captureEvents()` объекта Window, Document или Layer.

См. также

Event, Window.captureEvents(); главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.onmouseout

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при выходе курсора мыши из элемента

Синтаксис

```
<элемент onmouseout="обработчик" ... >
элемент.onmouseout
```

Описание

Свойство `onmouseout` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь выводит курсор мыши из *элемента*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onmouseout` определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onmouseout`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *элемент*.

См. также

Event, Link.onmouseout; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.onmouseover

JavaScript 1.2; HTML 4.0

обработчик, вызываемый при перемещении курсора мыши над элементом

Синтаксис

```
<элемент onmouseover="обработчик" ... >
элемент.onmouseover
```

Описание

Свойство `onmouseover` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь перемещает курсор мыши над *элементом*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onmouseover` определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onmouseover`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *элемент*.

См. также

Event, Link.onmouseover; главу 19

HTMLElement.onmouseup

JavaScript 1.2; HTML 4.0

обработчик, вызываемый, когда пользователь отпускает кнопку мыши

Синтаксис

```
<элемент onmouseup="обработчик" ... >
элемент.onmouseup
```

Описание

Свойство `onmouseup` объекта HTMLElement задает функцию обработки события, вызываемую, когда пользователь отпускает кнопку мыши над *элементом*.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onmouseup` определяющего объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *элемента*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onmouseup`. В IE аргумент не передается, но соответствующий объект Event доступен как свойство `event` объекта Window, содержащего *элемент*.

В Netscape 4 свойство `Event.which` задает, какая кнопка мыши была нажата. В IE 4 номер кнопки задается свойством `Event.button`.

См. также

Event, HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

HTMLElement.removeAttribute()

Internet Explorer 4, Netscape 6

удаляет атрибут

Синтаксис

элемент.removeAttribute(*имя*)

Аргументы

имя – имя атрибута, который должен быть удален.

Возвращаемое значение

true в случае успеха; false в случае неудачи.

Описание

removeAttribute() удаляет атрибут *имя* из *элемента*. Если *элемент* не имеет атрибута с именем *имя*, этот метод возвращает false.

HTMLElement.scrollIntoView()

Internet Explorer 4

делает элемент видимым

Синтаксис

элемент.scrollIntoView(*top*)

Аргументы

top

Необязательный логический аргумент, указывающий, должен ли элемент прокручиваться к верхнему или к нижнему краю экрана. Если он равен true или опущен, *элемент* перемещается к верхнему краю экрана. Если он равен false, *элемент* перемещается к нижнему краю экрана.

Описание

scrollIntoView() прокручивает документ, содержащий *элемент*, так, чтобы верхний край *элемента* был выровнен по верхнему краю видимой области или чтобы нижний край *элемента* был выровнен по нижнему краю видимой области.

HTMLElement.setAttribute()

Internet Explorer 4, Netscape 6

устанавливает значение атрибута

Синтаксис

элемент.setAttribute(*имя*, *значение*)

Аргументы

имя

Имя устанавливаемого атрибута.

значение

Устанавливаемое значение.

Описание

setAttribute() устанавливает атрибут *имя элемента* равным *значению*.

Image

JavaScript 1.1

изображение в документе HTML **наследуется от HTMLElement**

Синтаксис

document.images[*i*]
document.images.length
document.*имя_изображения*

Конструктор

new Image(*ширина*, *высота*)

Аргументы

ширина, *высота*

Ширина и высота изображения; могут не задаваться.

Свойства

Image наследует свойства от HTMLElement и определяет следующие свойства, большинство из которых соответствуют атрибутам тега <img>. В JavaScript 1.1 и более поздних версиях свойства src и lowsrc доступны для чтения и записи и могут быть установлены для изменения отображаемого изображения. В броузерах, не допускающих переформатирование документа, таких как IE 3 и Netscape 4, прочие свойства доступны только для чтения.

border

Целое, задающее ширину границы вокруг изображения в пикселах. Значение этого свойства устанавливается с помощью атрибута border. Изображения имеют границы, только если они находятся внутри гиперссылок.

complete

Доступное только для чтения логическое значение, указывающее, загрузилось ли изображение целиком или, точнее, закончил ли броузер свои попытки загрузить изображение. Если при загрузке возникла ошибка или если загрузка прервалась, свойство complete все равно устанавливается в true.

height

Целое, задающее высоту изображения в пикселах. Значение этого свойства устанавливается атрибутом height.

hspace

Целое, задающее объем дополнительного горизонтального пространства в пикселах, вставляемого слева и справа от изображения. Это значение устанавливается атрибутом hspace.

lowsrc

Строка, доступная для чтения и для записи, задающая URL альтернативного изображения (обычно меньшего размера) для отображения, когда пользовательский броузер работает на мониторе низкого разрешения. Начальное значение этого свойства задается атрибутом lowsrc тега <img>.

Установка этого свойства не дает немедленного эффекта. Однако если устанавливается свойство src, загружается новое изображение и на системах с низким разрешением вместо только что измененного значения src используется текущее значение lowsrc.

name

Строковое значение, заданное атрибутом name, которое указывает имя изображения. Когда изображению присваивается имя с помощью атрибута HTML name, ссылка на это изображение, кроме помещения в массив document.images[], автоматически помещается в свойство *имя_изображения* документа.

src

Строка, доступная для чтения и для записи, задающая URL изображения, отображаемого в броузере. Начальное значение этого свойства задается атрибутом src тега <img>. Если установить это свойство равным URL нового изображения, броузер загрузит и отобразит новое изображение (или, в системе низкого разрешения, изображение, указанное в свойстве lowsrc). Это полезно для изменения внешнего вида страниц в ответ на действия пользователя, а также может применяться для создания простой анимации.

vspace

Целое, задающее объем дополнительного вертикального пространства в пикселах, вставляемое над и под изображением. Его значение устанавливается атрибутом vspace.

width

Целое, задающее ширину изображения в пикселах. Это значение устанавливается атрибутом width.

Обработчики событий

Изображение наследует обработчики событий от HTMLElement и определяет следующее:

onabort

Вызывается, если пользователь прерывает загрузку изображения.

onerror

Вызывается, если происходит ошибка при загрузке изображения.

onload

Вызывается, когда изображение успешно завершает загрузку.

Синтаксис HTML

Объект Image создается с помощью стандартного тега HTML <img>. Некоторые атрибуты <img> в приведенном синтаксисе отсутствуют, так как они недоступны в JavaScript:

```
<img src="url"                    // Показываемое изображение
     width="пикселы"              // Ширина изображения
     height="пикселы"             // Высота изображения
     [ name="имя_изображения" ]   // Имя свойства для изображения
     [ lowsrc="url" ]             // Альтернативное изображение в низком разрешении
     [ border="пикселы" ]         // Ширина границы изображения
     [ hspace="пикселы" ]         // Дополнительное горизонтальное пространство
                                  // вокруг изображения
     [ vspace="пикселы" ]         // Дополнительное вертикальное пространство
                                  // вокруг изображения
     [ onload="обработчик" ]      // Вызывается, когда изображение полностью загружено
     [ onerror="обработчик" ]     // Вызывается в случае ошибки загрузки
     [ onabort="обработчик" ]     // Вызывается, если пользователь прекращает загрузку
>
```

Описание

Объекты Image в массиве document.images[] представляют изображения, встраиваемые в документ HTML с помощью тега <img>. src – это наиболее интересное из свойств этих объетов; когда вы устанавливаете это свойство, броузер загружает и показывает изображение, заданное новым значением.

Объекты Image можно динамически создавать в коде JavaScript с помощью функции-конструктора Image(). Обратите внимание, что этот конструктор не имеет аргумента, задающего загружаемое изображение. Как и для изображений, созданных с помощью кода HTML, можно заставить броузер загрузить изображение, явно установив свойство src для любого изображения, созданного вами. Нет способа отобразить объект Image в веб-броузере. Можно лишь заставить объект Image загрузить изображение с помощью свойства src. Однако это может быть полезно, поскольку при этом изображение загружается в кэш броузера. Позднее, если тот же URL изображения указывается для одного из изображений в массиве images[], то оно, будучи загруженным заранее, отображается быстро. Это можно сделать посредством следующих строк кода:

```
document.images[2].src = preloaded_image.src;
document.toggle_image.src = toggle_off.src;
```

Применение

Установка свойства src объекта Image – средство реализации простой анимации на веб-страницах. Это также отличная технология для изменения графики на странице при взаимодействии пользователя с ней. Например, можно создать собственную кнопку Submit, используя изображение и гиперссылку. В начале кнопка содержит графическое изображение, представляющее отключенное состояние, и остается в таком виде, пока пользователь корректно не введет всю обязательную информацию. Тогда графическое изображение меняется, и пользователь может подтвердить форму.

Image.onabort

JavaScript 1.1

обработчик, вызываемый при прерывании загрузки изображения

Синтаксис

```
<img ... onabort="обработчик" ... >
image.onabort
```

Описание

Свойство onabort объекта Image задает функцию обработки события, вызываемую, когда пользователь прерывает загрузку изображения (например, нажав кнопку Stop).

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом onabort определяющего объект Image тега <img>. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *image*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onabort. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего *image*.

Image.onerror

JavaScript 1.1

обработчик, вызываемый при возникновении ошибки во время загрузки изображения

Синтаксис

```
<img ... onerror="обработчик" ... >
image.onerror
```

Описание

Свойство `onerror` объекта Image задает функцию обработки события, вызываемую, когда возникает ошибка при загрузке изображения.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onerror` определяющего объект Image тега `<img>`. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *image*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onerror`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *image*.

Image.onload

JavaScript 1.1

обработчик, вызываемый по завершении загрузки изображения

Синтаксис

```
<img ... onload="обработчик" ... >
image.onload
```

Описание

Свойство `onload` объекта Image задает функцию обработки события, которая вызывается, когда изображение загружено успешно.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onload` определяющего объект Image тега `<img>`. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия *image*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onabort`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего *image*.

См. также

Главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Input

JavaScript 1.0; расширен в JavaScript 1.1

элемент ввода в форме HTML **наследуется от HTMLElement**

Синтаксис

```
form.elements[i]
form.name
```

Свойства

Input наследует свойства от HTMLElement и определяет или перекрывает следующие:

`checked`

Доступное для чтения и записи логическое значение, которое определяет, отмечен ли в данный момент элемент формы Checkbox или Radio. Состояние этих элементов-кнопок можно установить, задав значение данного свойства. Это свойство не используется другими элементами формы.

`defaultChmecked`

Доступное только для чтения логическое значение, определяющее, отмечен ли элемент Checkbox или Radio по умолчанию. Это свойство предназначено для восстановления элемента Checkbox или Radio к его значению по умолчанию, когда значение сбрасывается, и не имеет смысла для других элементов форм. `defaultChecked` соответствует атрибуту `checked` тега HTML `<input>`, создавшего элемент формы. Если `checked` присутствовал в теге, то свойство `defaultChecked` равно `true`. В противном случае `defaultChecked` равно `false`.

`defaultValue`

Задает первоначальный текст, отображаемый в элементе формы, и значение, к которому восстанавливается элемент при сбросе формы. Это свойство применяется только элементами Text, Textarea и Password. По соображениям безопасности оно не используется элементом FileUpload. Для элементов Checkbox и Radio эквивалентным свойством является `defaultChecked`.

`form`

Доступная только для чтения ссылка на объект Form, содержащий данный элемент. Свойство `form` позволяет обработчикам событий одного элемента формы легко обращаться к смежным элементам в той же форме. Когда вызывается обработчик события, ключевое слово `this` ссылается на элемент формы, для которого был вызван обработчик. Следовательно, обработчик события может обратиться к содержащей его форме посредством выражения `this.form` и, следовательно, может обращаться к смежным элементам по имени или используя массив `elements[]` объекта Form для обращения к ним по номерам.

`length`

Для элемента формы Select это свойство задает количество опций или вариантов выбора (каждый из которых представлен объектом Option), содержащихся в массиве `options[]` элемента. См. справочную страницу «Select».

`name`

Строка, доступная только для чтения, которая задается атрибутом `name`, определяющая имя этого элемента. Это имя может использоваться для обращения к элементу, как показано в разделе «Синтаксис». См. справочную страницу «Input.name».

`options[]`

Для элемента формы Select этот массив содержит объекты Option, представляющие варианты выбора, отображаемые объектом Select. Количество элементов в массиве задается с помощью свойства `length` элемента Select. См. справочную страницу «Select».

`selectedIndex`

Для элемента формы Select это целое значение определяет, какая опция из отображаемых в объекте Select выбрана в данный момент. В JavaScript 1.1 это свойство доступно для чтения и для записи. В JavaScript 1.0 оно доступно только для чтения. См. справочную страницу «Select».

`type` *[JavaScript 1.1]*

Это строка, доступная только для чтения и определяющая тип элемента формы. См. справочную страницу «Input.type».

`value`

Представляет собой строку, определяющую значение, отображаемое элементом, и/или строку, которая должна отправляться на сервер при передаче содержащей его формы. См. справочную страницу «Input.value».

Методы

Input наследует методы от HTMLElement и определяет или перекрывает следующее:

`blur()`

Убирает клавиатурный фокус из элемента.

`click()`

Имитирует нажатие клавиши мыши на элементе формы.

`focus()`

Передает элементу клавиатурный фокус.

`select()`

Для элементов форм, отображающих редактируемый текст, выбирает текст, присутствующий в элементе.

Обработчики событий

Input наследует обработчики событий от HTMLElement и определяет или перекрывает следующие:

`onblur`

Вызывается, когда пользователь выводит клавиатурный фокус из элемента.

`onchange`

Для элементов форм, не являющихся кнопками, этот обработчик события вызывается, когда пользователь вводит или выбирает новое значение.

`onclick`

Для элементов форм, являющихся кнопками, этот обработчик события вызывается, когда пользователь щелкает по кнопке.

`onfocus`

Вызывается, когда пользователь передает элементу клавиатурный фокус.

Описание

Элементы формы хранятся в массиве `elements[]` объекта Form. Содержимым этого массива являются объекты Input, представляющие индивидуальные кнопки, поля ввода и другие управляющие элементы, присутствующие в форме. Многие типы элементов ввода создаются с помощью тега `<input>`; другие создаются с помощью тегов `<select>`, `<option>` и `<textarea>`. Различные элементы ввода формы имеют довольно

много общих свойств, методов и обработчиков события, описанных на этой справочной странице. Особенности конкретных типов элементов формы описываются на их собственных страницах.

Объект Input определяет много общих свойств, методов и обработчиков событий, но не все они разделяются всеми типами элементов формы. Например, объект Button вызывает обработчик события `onclick`, но не обработчик `onchange`, а объект Text вызывает `onchange`, но не вызывает `onclick`. Следующий рисунок показывает все элементы форм и связанные с ними свойства.

ЭЛЕМЕНТ \ СВОЙСТВО	checked	defaultChecked	defaultValue	form	length	name	options	selectedIndex	type	value	blur()	click()	focus()	select()	onblur	onchange	onclick	onfocus
Button				•		•			•	•	•	•	•		•		•	•
Checkbox	•	•		•		•			•	•	•	•	•		•		•	•
Radio	•	•		•		•			•	•	•	•	•		•		•	•
Reset				•		•			•	•	•	•	•		•		•	•
Submit				•		•			•	•	•	•	•		•		•	•
Text			•	•		•			•	•	•		•	•	•	•		•
Textarea			•	•		•			•	•	•		•	•	•	•		•
Password			•	•		•			•	•	•		•	•	•	•		•
FileUpload			•	•		•			•	•	•		•	•	•	•		•
Select				•	•	•	•	•	•		•	•	•		•	•		•
Hidden				•		•			•	•								

Имеется две основных категории элементов форм. Первая – это кнопки: Button, Checkbox, Radio, Reset и Submit. В этих элементах имеется обработчик события `onclick`, но нет обработчика `onchange`. Кроме того, они реагируют на метод `click()`, но не реагируют на `select()`. Вторая категория включает элементы, отображающие текст: Text, Textarea, Password и FileUpload. Эти элементы имеют обработчик события `onchange`, но не `onclick`, и реагируют на метод `select()`, но не на `click()`.

Элемент Select – это особый случай. Он создается с помощью тега `<select>` и меньше похож на элементы `<input>`, чем другие элементы формы. Несмотря на то что элемент Select формально представляется другим типом объекта, все равно удобно рассматривать его как объект Input.

См. также

Button, Checkbox, FileUpload, Form, Hidden, Password, Radio, Reset, Select, Submit, Text, Textarea; главу 15; HTMLInputElement в «Справочнике по W3C DOM»

Input.blur() JavaScript 1.0

убирает клавиатурный фокус ввода из элемента формы

Синтаксис

input.blur()

Описание

Метод `blur()` элемента формы убирает клавиатурный фокус из этого элемента, не вызывая обработчик события `onblur`; по существу, он противоположен методу `focus()`. Однако метод `blur()` никуда не передает клавиатурный фокус, вот почему вызов этого метода полезен в единственном случае – непосредственно перед тем, как вы передаете клавиатурный фокус другому элементу с помощью метода `focus()`, но не хотите вызывать обработчик события `onblur`. То есть после явного выведения фокуса из элемента вы не будете уведомлены, когда выведете его неявно, вызвав метод `focus()` для другого элемента.

Все элементы формы, отличные от Hidden, поддерживают метод `blur()`. К сожалению, не все платформы одинаково хорошо поддерживают клавиатурную навигацию. В Netscape 2 и 3 для платформ Unix метод `blur()` работает только для тех элементов формы, которые отображают текст: Text, Textarea, Password и FileUpload.

Input.click() JavaScript 1.0

имитирует нажатие кнопки мыши на элементе формы

Синтаксис

input.click()

Описание

Метод `click()` элемента формы имитирует нажатие кнопки мыши на элементе формы, но не вызывает обработчик события `onclick` для элемента.

Метод `click()` не часто бывает полезен. Он не вызывает обработчик события `onclick`, поэтому нет смысла вызывать его для элементов Button – они не делают ничего кроме того, что определено обработчиком `onclick`. Вызов `click()` для элемента Submit или Reset передает данные или очищает форму, но это достигается и более непосредственным способом – с помощью методов `submit()` и `reset()` самого объекта Form.

Input.focus() JavaScript 1.0

дает клавиатурный фокус элементу формы

Синтаксис

input.focus()

Описание

Метод `focus()` элемента формы передает клавиатурный фокус элементу, не вызывая обработчик события `onfocus`. То есть он делает элемент активным для клавиатурной навигации и клавиатурного ввода. Другими словами, если вы вызовете `focus()` для элемента Text, то любой текст, набранный пользователем, появляется в этом текстовом элементе. А если вы вызовете `focus()` для элемента Button, пользователь сможет активизировать эту кнопку с клавиатуры.

Все элементы формы, отличные от Hidden, поддерживают метод `focus()`. К сожалению, не все платформы одинаково хорошо поддерживают клавиатурную навигацию. В Netscape 2 и 3 на платформах Unix метод `focus()` работает только для тех элементов формы, которые отображают текст: Text, Textarea, Password и FileUpload.

Input.name

JavaScript 1.0

имя элемента формы

Синтаксис

input.name

Описание

name – это доступное только для чтения строковое свойство любого элемента формы. Значение этого свойства устанавливается атрибутами name тега HTML <input>, определяющего элемент формы.

Имя элемента формы используется для двух целей. Во-первых, при передаче данных формы. Данные каждого элемента формы обычно отправляются в формате:

имя=значение

где *имя* и *значение* при необходимости кодируются для передачи. Если имя для элемента формы не задано, данные этого элемента не могут быть переданы на веб-сервер.

Во-вторых, свойство name применяется при обращении к элементу формы в коде JavaScript. Значение этого свойства – это ссылка на элемент. Например, если address – это форма, содержащая текстовый элемент ввода с именем zip, то address.zip будет ссылаться на этот элемент ввода.

Для элементов формы Radio и Checkbox принято определять несколько связанных объектов, имеющих общее значение свойства name. В этом случае данные передаются на сервер в следующем формате:

имя=значение1, значение2, ..., значениеn

Аналогично, в JavaScript все элементы, разделяющие это имя, становятся элементами массива с этим массивом. То есть если четыре объекта Checkbox в форме order разделяют общее имя options, они будут доступны в JavaScript как элементы массива order.options[].

Input.onblur

JavaScript 1.0

обработчик, вызываемый при потере фокуса элементом формы

Синтаксис

<input type="*type*" onblur="*обработчик*">
input.onblur

Описание

Свойство onblur объекта Input определяет функцию обработки события, вызываемую, когда пользователь выводит клавиатурный фокус из элемента ввода. Вызов blur() для выведения фокуса из элемента не приводит к вызову onblur для этого объекта. Заметьте, однако, что вызов focus() для передачи фокуса какому-либо другому элементу приводит к вызову обработчика события onblur для того элемента, который в данный момент имеет фокус.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом onblur определяющего данный объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия объекта *input*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onblur`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего элемент *`input`*.

Обработчик события `onblur` доступен для всех элементов формы, кроме элемента Hidden. Однако в Netscape на платформах Unix он вызывается только для элементов ввода текста: Text, Textarea, Password и FileUpload. Заметьте, что в JavaScript 1.1 объект Window также определяет обработчик события `onblur`.

См. также

Window.onblur; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Input.onchange

JavaScript 1.0

обработчик, вызываемый при изменении значения элемента формы

Синтаксис

```
<input type="type" onchange="обработчик">
input.onchange
```

Описание

Свойство `onchange` объекта Input определяет функцию обработки события, которая вызывается, когда пользователь изменяет значение, отображаемое элементом формы. Таким изменением может быть редактирование текста, выводимого в элементах Text, Textarea, Password или FileUpload, или выбор или отмена выбора опции в элементе Select. Заметьте, что этот обработчик события вызывается, только когда пользователь завершает это изменение – он не вызывается, если отображаемое элементом значение изменяется программой JavaScript.

Также заметьте, что обработчик `onchange` не вызывается при каждом вводе или удалении символа в элементе ввода текста в форме. `onchange` не предназначен для обработки событий на посимвольной основе. Вместо этого `onchange` вызывается, когда пользователь завершает редактирование. Броузер предполагает, что редактирование завершено, когда клавиатурный фокус перемещается от текущего элемента к какому-либо другому элементу – например, когда пользователь щелкает по следующему элементу формы. Уведомления о событиях на посимвольной основе рассматриваются на справочной странице «HTMLElement.onkeypress».

Обработчик события `onchange` не используется элементом Hidden и ни одним из элементов-кнопок. Эти элементы – Button, Checkbox, Radio, Reset и Submit – используют вместо этого обработчик события `onclick`.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onchange` определяющего данный объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия объекта *`input`*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onchange`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего элемент *`input`*.

См. также

HTMLElement.onkeypress; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Input.onclick

JavaScript 1.0; расширен в JavaScript 1.1

обработчик, вызываемый при щелчке по элементу формы

Синтаксис

```
<input type="type" onclick="обработчик">
input.onclick
```

Описание

Свойство `onclick` объекта Input определяет функцию обработки события, которая вызывается, когда пользователь щелкает по элементу ввода. Он не вызывается, когда для элемента вызывается метод `click()`.

Только элементы форм, являющиеся кнопками, возбуждают обработчик события `onclick`. Это элементы Button, Checkbox, Radio, Reset и Submit. Другие элементы форм используют вместо `onclick` обработчик события `onchange`.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onclick` определяющего данный объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия объекта *input*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onclick`. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства `event` объекта Window, содержащего элемент *input*.

Обратите внимание, что элементы Reset и Submit при щелчке выполняют действие по умолчанию: они, соответственно, сбрасывают и передают форму, в которой они содержатся. При помощи обработчиков события `onclick` каждого из этих элементов можно осуществлять действия, дополняющие эти действия по умолчанию. В JavaScript 1.1 можно также предотвратить выполнение этих действий по умолчанию, вернув `false`. То есть если обработчик `onclick` кнопки Reset возвращает `false`, форма не очищается, а если обработчик `onclick` кнопки Submit возвращает `false`, форма не передается. Обратите внимание, что то же самое может быть сделано с помощью обработчиков события `onsubmit` и `onreset` самого объекта Form.

И наконец, обратите внимание, что объект Link также определяет обработчик события `onclick`.

См. также

Link.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Input.onfocus

JavaScript 1.0

обработчик, вызываемый при получении фокуса элементом формы

Синтаксис

```
<input type="type" onfocus="обработчик">
input.onfocus
```

Описание

Свойство `onfocus` объекта Input определяет функцию обработки события, вызываемую, когда пользователь переводит клавиатурный фокус к этому элементу ввода. Вызов метода `focus()` для установки фокуса на элемент не приводит к возбуждению события `onfocus` для этого объекта.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, которые задаются атрибутом `onfocus` определяющего данный объект тега HTML. Если функция обработки события определяется атрибутом HTML, то она исполняется в области действия объекта *`input`*, а не в области действия содержащего его окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику `onfocus`. В IE аргумент не передается, но соответствующий объект Event доступен как свойство `event` объекта Window, содержащего элемент *`input`*.

Обработчик события `onfocus` доступен для всех элементов формы, кроме элемента Hidden. Однако в Netscape на платформах Unix он вызывается только для элементов ввода текста: Text, Textarea, Password и FileUpload. Заметьте, что в JavaScript 1.1 объект Window также определяет обработчик события `onfocus`.

См. также

Window.onfocus; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Input.select() JavaScript 1.0

выбирает текст в элементе формы

Синтаксис

`input``.select()`

Описание

Метод `select()` выделяет текст, отображаемый в элементе Text, Textarea, Password или FileUpload. На разных платформах это может выглядеть по-разному, но обычно вызов этого метода дает тот же результат, что и перетаскивание мыши над текстом в указанном объекте Text. На большинстве платформ возникают следующие эффекты:

- Текст выделяется, часто отображаясь в инвертированных цветах.
- Если текст остается выбранным, то когда пользователь введет символ в следующий раз, выбранный текст будет заменен только что введенным символом.
- На некоторых платформах текст становится доступным для операций вставки-удаления.

Пользователь обычно отменяет выбор текста щелчком по объекту Text или перемещением курсора. После отмены выбора пользователь может добавлять и удалять отдельные символы, не заменяя текстовое значение целиком.

Input.type JavaScript 1.1

тип элемента формы

Синтаксис

`input``.type`

Описание

type – это доступное только для чтения строковое свойство всех элементов формы, задающее тип элемента. Значения этого свойства для всех возможных элементов формы дано в следующей таблице.

Тип объекта	Тег HTML	Свойство type
Button	<input type="button">	"button"
Checkbox	<input type="checkbox">	"checkbox"
FileUpload	<input type="file">	"file"
Hidden	<input type="hidden">	"hidden"
Password	<input type="password">	"password"
Radio	<input type="radio">	"radio"
Reset	<input type="reset">	"reset"
Select	<select>	"select-one"
Select	<select multiple>	"select-multiple"
Submit	<input type="submit">	"submit"
Text	<input type="text">	"text"
Textarea	<textarea>	"textarea"

Заметьте, что элемент Select имеет два возможных значения типа, в зависимости от того, допускает ли он одинарный или множественный выбор. Заметьте также, что в отличие от других свойств элементов, type не доступен в JavaScript 1.0.

Input.value

Netscape 2; имеет ошибки в Internet Explorer 3

значение, отображаемое или отсылаемое элементом формы

Синтаксис

input.value

Описание

value – это доступное для чтения и для записи строковое свойство всех элементов формы, определяющее значение, которое отображается элементом формы и/или отсылается элементом, когда форма передается. Например, свойство value элемента – это введенные пользователем данные, которые также являются значением, отсылаемым с формой. С другой стороны, для объекта Checkbox свойство value задает строку, которая не отображается, но отсылается с формой, если элемент Checkbox – тот, что был отмечен при передаче данных формы.

Начальное значение свойства value задается атрибутом value тега HTML, определяющего элемент формы.

Для объектов Button, Submit и Reset свойство value задает текст, появляющийся внутри кнопки. На некоторых платформах изменение свойства value этих элементов реально изменяет текст, показываемый кнопками на экране. Однако на некоторых платформах это не работает, и я бы не рекомендовал этот прием. Изменение метки кнопки может изменить размер кнопки, приведя к перекрытию и скрыванию других частей документа.

Элемент Select имеет свойство value, как и другие элементы формы, но не использует его. Вместо этого значение, отсылаемое этим элементом, задается свойствами value содержащихся в нем объектов Option.

По соображениям безопасности свойство value элемента FileUpload доступно только для чтения.

JavaArray

Netscape 3 LiveConnect

представление в JavaScript массива Java

Синтаксис

```
javaarray.length  // Длина массива
javaarray[index]  // Читает или пишет элемент массива
```

Свойства

length

Доступное только для чтения целое значение, задающее количество элементов в массиве Java, представляемом объектом JavaArray.

Описание

Объект JavaArray – это представление в JavaScript массива Java, позволяющее коду на JavaScript читать и писать элементы массива, используя знакомый вам синтаксис массивов JavaScript. Кроме того, объект JavaArray имеет поле length, определяющее число элементов в массиве Java.

При чтении и записи значений элементов массива преобразование данных между представлениями JavaScript и Java осуществляется системой автоматически. Подробности см. в главе 22.

Применение

Обратите внимание, что массивы Java отличают от массивов JavaScript два важных момента. Во-первых, массивы Java имеют фиксированную длину, задаваемую при их создании. По этой причине поле length в JavaArray доступно только для чтения. Во-вторых, массивы Java являются *типизированными* (то есть все их элементы должны иметь одинаковый тип данных). Попытка присвоить значение неверного типа элементу массива приводит к ошибке или исключению JavaScript.

Пример

java.awt.Polygon – это объект JavaClass. Мы можем создать JavaObject, представляющий экземпляр этого класса:

```
p = new java.awt.Polygon();
```

Объект p имеет свойства xpoints и ypoints, являющиеся объектами JavaArray, которые представляют массивы Java, состоящие из целых значений. Содержимое этих массивов можно проинициализировать посредством следующего кода JavaScript:

```
for(int i = 0; i < p.xpoints.length; i++)
    p.xpoints[i] = Math.round(Math.random()*100);
for(int i = 0; i < p.ypoints.length; i++)
    p.ypoints[i] = Math.round(Math.random()*100);
```

См. также

getClass(), JavaClass, JavaObject, JavaPackage, свойство `java` объекта Window; главу 22

JavaClass

Netscape 3 LiveConnect

JavaScript-представление Java-класса

Синтаксис

```
javaclass.статический_член     // Чтение или запись в статическое поле
                               // или вызов статического метода Java
new javaclass(...)             // Создает новый Java-объект
```

Свойства

Каждый объект JavaClass содержит свойства, имеющие те же имена, что и открытые статические поля и методы представляемого им Java-класса. Эти свойства позволяют читать и писать значения статических полей класса и вызывать статические поля класса. Все объекты JavaClass имеют разные свойства; перечислить свойства любого заданного объекта JavaClass можно посредством цикла `for/in`.

Описание

Объект JavaClass является JavaScript-представлением Java-класса. Свойства объекта JavaClass представляют открытые статические поля и методы (иногда называемые полями классов и методами классов) для представляемого им класса. Обратите внимание, что объект JavaClass не имеет свойств, представляющих поля *экземпляра* Java-класса – индивидуальные экземпляры Java-класса представлены объектами JavaObject.

Объект JavaClass реализует функциональность LiveConnect, позволяющую программам JavaScript читать и писать статические переменные Java-классов, используя обычный синтаксис JavaScript. Он также предоставляет функциональность, позволяющую JavaScript вызывать статические методы Java-класса.

Кроме создания для JavaScript возможности читать и писать значения переменных и методов Java, объект JavaClass позволяет программам JavaScript создавать Java-объекты (представляемые объектом JavaObject), используя ключевое слово `new` и метод-конструктор JavaClass.

Преобразование данных, необходимое для взаимодействия между JavaScript и Java через объект JavaClass, осуществляется автоматически LiveConnect (см. главу 22).

Применение

Имейте в виду, что Java – это *типизированный* язык. Это значит, что каждое из полей объекта имеет определенный тип данных и ему могут быть присвоены только данные этого типа. Попытка присвоить полю значение неверного типа приведет к ошибке JavaScript или к исключению. Попытка вызвать метод с аргументами неверного типа также приведет к ошибке или исключению.

Пример

`java.lang.System` – это объект JavaClass, представляющий класс *java.lang.System* в Java. Статическое поле этого класса можно прочитать посредством следующего кода:

```
var java_console = java.lang.System.out;
```

А так можно вызвать статический метод этого класса:

```
var version = java.lang.System.getProperty("java.version");
```

И наконец, объект JavaClass также позволяет создавать новые объекты Java:

```
var java_date = new java.lang.Date();
```

См. также

getClass(), JavaArray, JavaObject, JavaPackage, свойство `java` объекта Window; главу 22

JavaObject

Netscape 3 LiveConnect

JavaScript-представление Java-объекта

Синтаксис

```
javaobject.член     // Чтение или запись поля или вызов метода экземпляра
```

Свойства

Каждый объект JavaObject содержит свойства, имеющие те же имена, что и открытые поля и методы экземпляра (но не статические поля и методы класса) Java-объекта, который он представляет. Эти свойства позволяют читать и писать значения открытых свойств и вызывать открытые методы. Свойства данного объекта JavaObject, очевидно, зависят от типа представляемого им объекта Java. В цикле `for/in` можно перечислить свойства любого заданного JavaObject.

Описание

Объект JavaObject – это JavaScript-представление Java-объекта. Свойства объекта JavaObject представляют открытые поля экземпляра и открытые методы экземпляра, определенные в объекте Java. (Поля и методы класса представлены объектом JavaClass.)

Объект JavaObject реализует функциональность LiveConnect, позволяющую программам JavaScript читать и писать открытые поля экземпляра Java-объекта, используя обычный синтаксис JavaScript. Он также предоставляет функциональность, позволяющую JavaScript вызывать метод Java-объекта. Преобразования данных между представлениями в JavaScript и Java осуществляются LiveConnect автоматически. Подробности см. в главе 22.

Применение

Имейте в виду, что Java – это *типизированный* язык. Это значит, что каждое поле объекта имеет определенный тип данных, и вы можете установить его равным только значениям этого типа. Например, поле `width` объекта `java.awt.Rectangle` – это целое поле, и попытка присвоить ему строку приведет к ошибке или исключению JavaScript.

Пример

`java.awt.Rectangle` – это JavaClass, представляющий класс *java.awt.Rectangle*. Мы можем создать JavaObject, представляющий экземпляр этого класса, так:

```
var r = new java.awt.Rectangle(0,0,4,5);
```

Затем мы можем прочитать открытые переменные этого экземпляра JavaObject `r` с помощью следующего кода:

```
var perimeter = 2*r.width + 2*r.height;
```

Мы также можем установить значения открытых переменных экземпляра `r`, используя синтаксис JavaScript:

```
r.width = perimeter/4;
r.height = perimeter/4;
```

См. также

getClass(), JavaArray, JavaClass, JavaPackage, свойство `java` объекта Window; главу 22

JavaPackage

Netscape 3 LiveConnect

JavaScript-представление Java-пакета

Синтаксис

```
package.имя_пакета     // Ссылается на другой JavaPackage
package.имя_класса     // Ссылается на объект JavaClass
```

Свойства

Свойства объекта JavaPackage – это имена содержащихся в нем объектов JavaPackage и объектов JavaClass. Эти свойства различны для каждого индивидуального объекта JavaPackage. Обратите внимание, что невозможно обойти список имен свойств объекта Package в цикле `for/in` JavaScript. Для определения пакетов и классов, содержащихся в каждом конкретном пакете, обращайтесь к справочному руководству по Java.

Описание

Объект JavaPackage – это JavaScript-представление Java-пакета. Пакет в Java – это коллекция связанных классов. В JavaScript JavaPackage может содержать классы (представленные объектом JavaClass) и другие объекты JavaPackage.

В объекте Window имеются свойства `java`, `netscape` и `sun`, представляющие иерархии пакетов *java.**, *netscape.** и *sun.**. Эти объекты JavaPackage определяют свойства, которые ссылаются на другие объекты JavaPackage. Например, `java.lang` и `java.net` ссылаются на пакеты *java.lang* и *java.net*. JavaPackage `java.awt` содержит свойства с именами `Frame` и `Button`, которые оба ссылаются на объекты JavaClass и представляют классы *java.awt.Frame* и *java.awt.Button*.

Объект Window также определяет свойство с именем `Packages` – корневое свойство JavaPackage, свойства которого ссылаются на корни всех известных иерархий пакетов. Например, выражение `Packages.java.awt` – это то же самое, что `java.awt`.

Нет возможности использовать цикл `for/in` для определения имен пакетов и классов, содержащихся в JavaPackage. Эту информацию необходимо знать заранее. Ее можно найти в любом справочном руководстве по Java или изучив иерархию Java-классов.

Работа с пакетами, классами и объектами подробно рассмотрена в главе 22.

См. также

JavaArray, JavaClass, JavaObject; свойства `java`, `netscape`, `sun` и `Packages` объекта Window; главу 22

JSObject

Netscape 3, Internet Explorer 4

Java-представление объекта JavaScript

Синтаксис

```
public final class netscape.javascript.JSObject extends Object
```

Методы

`call()`

Вызывает метод объекта JavaScript.

`eval()`

Вычисляет строку JavaScript-кода в контексте объекта JavaScript.

`getMember()`

Получает значение свойства объекта JavaScript.

`getSlot()`

Получает значение элемента массива объекта JavaScript.

`getWindow()`

Получает «корневой» JSObject, представляющий JavaScript-объект Window веб-броузера.

`removeMember()`

Удаляет свойство из объекта JavaScript.

`setMember()`

Устанавливает значение свойства объекта JavaScript.

`setSlomt()`

Устанавливает значение элемента массива в объекте JavaScript.

`toString()`

Вызывает метод `toString()` объекта JavaScript и возвращает его результат.

Описание

JSObject – это класс Java, а не объект JavaScript; его нельзя применять в программах JavaScript. Но JSObject используется Java-апплетами, которые желают взаимодействовать с JavaScript путем чтения и записи свойств объектов и элементов массивов JavaScript, вызова методов JavaScript, вычисления и исполнения произвольных строк кода JavaScript. Поскольку JSObject представляет собой класс Java, необходимо разбираться в программировании Java, чтобы его использовать.

Подробную информацию о программировании с JSObject можно найти в главе 22.

См. также

Главу 22

JSObject.call()

Netscape 3, Internet Explorer 4

вызывает метод объекта JavaScript

Синтаксис

```
public Object call(String имя_метода, Object args[])
```

Аргументы

имя_метода

Имя метода JavaScript, который должен быть вызван.

args[]

Массив объектов Java, которые должны быть переданы методу как аргументы.

Возвращаемое значение

Объект Java, представляющий возвращаемое значение метода JavaScript.

Описание

Метод `call()` Java-класса JSObject вызывает именованный метод объекта JavaScript, представленного JSObject. Аргументы передаются методу в виде массива объектов Java, а возвращаемое значение JavaScript-метода возвращается в виде Java-объекта.

В главе 22 описано преобразование данных для аргументов метода из Java-объектов в значения JavaScript и возвращаемого значения из значения JavaScript в Java-объект.

JSObject.eval()

Netscape 3, Internet Explorer 4

вычисляет строку кода JavaScript

Синтаксис

```
public Object eval(String s)
```

Аргументы

s – строка, содержащая произвольные инструкции JavaScript, разделенные точками с запятой.

Возвращаемое значение

Значение JavaScript последнего вычисленного выражения в *s*, преобразованное затем в объект Java.

Описание

Метод `eval()` Java-класса JSObject вычисляет код JavaScript, содержащийся в строке *s*, в контексте объекта JavaScript, определяемого JSObject. Поведение метода `eval()` Java-класса JSObject во многом схоже с глобальной функцией JavaScript `eval()`.

Аргумент *s* может содержать любое число инструкций JavaScript, разделенных точками с запятой; эти инструкции исполняются в том порядке, в котором они указаны. Значение, возвращаемое `eval()` – это значение последней вычисленной инструкции или выражения в *s*.

JSObject.getMember()

Netscape 3, Internet Explorer 4

читает свойство объекта JavaScript

Синтаксис

```
public Object getMember(String имя)
```

Аргументы

имя – имя свойства, которое должно быть прочитано.

Возвращаемое значение

Объект Java, содержащий значение именованного свойства указанного JSObject.

Описание

Метод `getMember()` Java-класса JSObject читает и передает в Java значение указанного свойства объекта JavaScript. Возвращаемое значение может быть другим объектом JSObject или объектом Double, Boolean или String, но возвращается в виде обобщенного объекта Object, который необходимо преобразовать так, как это требуется.

JSObject.getSlot()

Netscape 3, Internet Explorer 4

читает элемент массива объекта JavaScript

Синтаксис

```
public Object getSlot(int индекс)
```

Аргументы

индекс

Индекс элемента массива, который должен быть прочитан.

Возвращаемое значение

Значение элемента массива по указанному *индексу* в объекте JavaScript.

Описание

Метод `getSlot()` Java-класса JSObject читает и возвращает в Java значение элемента массива по указанному *индексу* в объекте JavaScript. Возвращаемое значение может быть другим объектом JSObject или объектом Double, Boolean или String, но возвращается в виде обобщенного объекта Object, который необходимо преобразовать так, как это требуется.

JSObject.getWindow()

Netscape 3, Internet Explorer 4

возвращает начальный объект JSObject для окна броузера

Синтаксис

```
public static JSObject getWindow(java.applet.Applet апплет)
```

Аргументы

апплет

Объект Applet, работающий в окне веб-броузера, для которого должен быть получен JSObject.

Возвращаемое значение

JSObject, представляющий JavaScript-объект Window для окна веб-броузера, которое содержит указанный *апплет*.

Описание

Метод `getWindow()` — это первый метод JSObject, который вызывает любой Java-апплет. JSObject не определяет конструктора, а статический метод `getWindow()` предоставляет единственный способ получить начальный «корневой» JSObject, от которого могут быть получены другие объекты JSObject.

JSObject.removeMember()

Netscape 3, Internet Explorer 4

удаляет свойство объекта JavaScript

Синтаксис

```
public void removeMember(String имя)
```

Аргументы

имя – имя свойства, которое должно быть удалено из JSObject.

Описание

Метод `removeMember()` Java-класса JSObject удаляет указанное свойство из объекта JavaScript, представленного объектом JSObject.

JSObject.setMember()

Netscape 3, Internet Explorer 4

устанавливает значение свойства объекта JavaScript

Синтаксис

```
public void setMember(String имя, Object значение)
```

Аргументы

имя

Имя устанавливаемого в JSObject свойства.

значение

Значение, которое должно быть присвоено указанному свойству.

Описание

Метод `setMember()` Java-класса JSObject устанавливает значение указанного свойства объекта JavaScript из Java. Указанное *значение* может быть любым Java-объектом. Элементарные значения Java не могут передаваться этому методу. В JavaScript указанное *значение* доступно как объект JavaObject.

JSObject.setSlot()

Netscape 3, Internet Explorer 4

устанавливает значение элемента массива в объекте JavaScript

Синтаксис

```
public void setSlot(int индекс, Object значение)
```

Аргументы

индекс

Индекс элемента массива, который должен быть установлен в JSObject.

значение

Значение, которое должно быть присвоено указанному элементу массива.

Описание

Метод `setSlot()` Java-класса JSObject устанавливает значение элемента массива с указанным номером в объекте JavaScript из Java. Указанное *значение* может быть любым объектом Java. Элементарные значения Java не могут быть переданы этому методу. В JavaScript указанное *значение* доступно в виде объекта JavaObject.

JSObject.toString()

Netscape 3, Internet Explorer 4

возвращает строковое значение объекта JavaScript

Синтаксис

```
public String toString()
```

Возвращаемое значение

Строка, получаемая путем вызова метода `toString()` объекта JavaScript, представленного указанным Java-объектом JSObject.

Описание

Метод `toString()` Java-класса JSObject вызывает метод `toString()` объекта JavaScript, представленного объектом JSObject, и возвращает результат этого метода.

Layer

только Netscape 4; в Netscape 6 не поддерживается

независимый слой в документе DHTML

Синтаксис

```
document.layers[i]
```

Конструктор

```
new Layer(ширина, родитель)
```

Аргументы

ширина

Ширина нового слоя в пикселах.

родитель

Объект Layer или Window, который должен быть родителем этого создаваемого слоя. Этот аргумент может отсутствовать, в этом случае новый слой становится дочерним элементом текущего окна.

Замечания

Конструктор `Layer()` создает новый объект Layer и возвращает его. Можно установить его размер, положение и другие атрибуты с помощью различных свойств и методов объекта Layer, описанных в следующем списке. В частности, можно установить свойство `hidden` в `false`, сделав новый слой видимым. Способы установки содержимого слоя рассмотрены в описаниях свойств `src` и метода `load()`. Альтернативный способ – динамическое генерирование содержимого слоя посредством записи в его свойство `document`.

Конструктор `Layer()` можно вызывать только по завершении загрузки текущего документа и всех его слоев.

Свойства

`above`

Доступное только для чтения свойство, ссылающееся на объект Layer, расположенный в стеке непосредственно над данным слоем. Если такого слоя нет, `above` равен `null`.

`background`

Объект Image, задающий изображение, выступающее в качестве фона слоя. Исходное значение этого свойства определяется атрибутом `background` тега `<layer>`. Можно изменить фоновое изображение, установив свойство `background.src`. Если оно установлено в `null`, никакое изображение не показывается, и вместо этого отображается цвет фона (задаваемый свойством `bgColor`).

`below`

Свойство, доступное только для чтения, которое ссылается на объект Layer, расположенный в стеке непосредственно под данным слоем. Если такого слоя нет, `below` равно `null`.

`bgColor`

Доступное для чтения и для записи строковое свойство, задающее цвет фона для слоя. Исходное значение свойства определяется атрибутом `bgcolor` тега `<layer>`. Обратите внимание, что *`layer`*`.background` имеет приоритет над *`layer`*`.bgColor`, поэтому цвет, заданный этим слоем, появляется, только если свойство `background.src` данного слоя равно `null`.

`clip.bottom`

Координата Y нижнего края области отсечения слоя относительно *`layer`*`.top`.

`clip.height`

Высота области отсечения слоя. Установка значения этого свойства также приводит к установке значения *`layer`*`.clip.bottom`.

`clip.left`

Задает координату X левого края области отсечения слоя. Это значение определяется относительно *`layer`*`.left`.

`clip.right`

Задает координату X правого края области отсечения слоя. Это значение определяется относительно *`layer`*`.left`.

`clip.top`

Задает координату Y верхнего края области отсечения слоя. Это значение определяется относительно *`layer`*`.top`.

`clip.width`

Задает ширину области отсечения слоя. Установка значения этого свойства также приводит к установке значения *`layer`*`.clip.right`.

`document`

Доступная только для чтения ссылка на объект Document, содержащийся в слое.

`hidden`

Указывает, является ли слой скрытым (`true`) или видимым (`false`). Установка этого свойства равным `true` скрывает слой, а установка его в `false` делает слой видимым.

`layers[]`

Массив, который содержит любые дочерние объекты Layer данного слоя. То же самое, что и массив `document.layers[]` слоя.

`left`

Доступное для чтения и для записи целое значение, задающее координату X слоя относительно родительского слоя или документа. Установка этого свойства перемещает слой влево или вправо. `left` — это синоним `x`.

`name`

Строка, доступная для чтения и для записи, задающая имя слоя. Начальное значение этого свойства задается атрибутами `name` или `id` тега HTML, который использовался для создания слоя, и также выступает в качестве имени свойства объекта Document, ссылающегося на объект Layer.

`pageX, pageY`

Целые значения, доступные для чтения и для записи, которые задают координаты X и Y этого слоя относительно документа верхнего уровня. Обратите внимание, что эти координаты определяются относительно страницы верхнего уровня, а не относительно какого-либо родительского слоя.

`parentLayer`

Доступная только для чтения ссылка на объект Layer или Window, содержащий этот слой (являющийся для него родительским).

`siblingAbove, siblingBelow`

Ссылается на смежный объект Layer (то есть дочерний элемент того же родительского слоя), расположенный в стеке непосредственно над или под этим слоем. Если такого слоя нет, эти свойства равны `null`.

`srcм`

Строка, доступная для чтения и для записи, которая задает URL (если он есть) содержимого слоя. Присваивание этому свойству нового URL приводит к тому, что броузер читает содержимое этого URL и показывает его в слое. Заметьте, однако, что это не работает, пока текущий документ не будет проанализирован. По этой причине ннельзя устанавливать `src` в сценарии верхнего уровня. Устанавливать его надо в обработчике события или в функции, вызываемой из обработчика события.

`top`

Доступное для чтения и для записи целое значение, которое задает координату Y этого слоя относительно родительского слоя или документа. Установка значения этого свойства перемещает слой вверх или вниз. `top` — это синоним `y`.

`visibility`

Строка, доступная для чтения и для записи, которая определяет видимость слоя. Имеется три возможных допустимых значения: `"show"` обозначает, что слой должен быть видимым; `"hide"` — слой не должен быть видимым; `"inherit"` — слой должен наследовать видимость своего родительского слоя.

`window`

Ссылается на объект Window, содержащий слой, независимо от того, насколько глубоко слой вложен внутрь других слоев.

`x, y`

Координаты X и Y слоя относительно родительского слоя или документа. Установка этого свойства приводит к перемещению слоя. `x` — это синоним свойства `left`, а `y` — синоним свойства `top`.

`zIndex`

Задает позицию слоя в z-порядке, или в стеке слоев. Когда два слоя перекрываются, слой, имеющий большее значение `zIndex`, оказывается наверху и скрывает слой с меньшим значением `zIndex`. Если два смежных слоя имеют одинаковый

zIndex, слой, присутствующий в массиве layers[] родительского документа позднее, выводится позже и перекрывает присутствующий ранее слой.

zIndex – это свойство, доступное для чтения и записи. Его установка изменяет порядок в стеке и выводит слои в новом порядке. Установка этого свойства может привести к переупорядочиванию массива layers[] родительского документа.

Методы

captureEvents()

Задает типы событий, которые должны быть перехвачены.

handleEvent()

Передает событие соответствующему обработчику.

load()

Загружает новый URL и изменяет размер.

moveAbove()

Размещает один слой над другим.

moveBelow()

Размещает один слой под другим.

moveBy()

Перемещает слой на относительную позицию.

movemTo()

Перемещает слой на позицию относительно своего родительского слоя.

moveToAbsolute()

Перемещает слой на позицию относительно страницы.

offset()

Синоним moveBy().

releaseEvents()

Прекращает перехват указанных типов событий.

resizeBy()

Изменяет размер слоя на указанные значения.

resizeTo()

Изменяет размер слоя на указанный.

routeEvent()

Перенаправляет событие к следующему заинтересованному обработчику.

Синтаксис HTML

Объект Layer может быть создан с помощью специального тега HTML <layer> в Netscape:

```
<layer
   [ id="имя_слоя" ]      // Имя слоя
   [ left="x" ]           // Позиция относительно родительского слоя
   [ top="y" ]
   [ pagex="x" ]          // Позиция относительно документа верхнего уровня
   [ pagey="y" ]
   [ width="w" ]          // Размер слоя
```

```
    [ height="h" ]
    [ src="url" ]                // URL содержимого слоя
    [ clip="x,y,w,h" ]           // Прямоугольник отсечения для слоя
    [ clip="w,h" ]               // Альтернативный синтаксис: x,y принимаются равными 0
    [ zindex="z" ]               // Порядок в стеке
    [ above="имя_слоя" ]         // Альтернативные способы указания порядка в стеке
    [ below="имя_слоя" ]
    [ visibility="vis" ]         // "show", "hide" или "inherit"
    [ bgcolor="color" ]          // Цвет фона слоя
    [ background="url" ]         // Фоновое изображение слоя
    [ onmouseover="обработчик" ] // Вызывается, когда курсор мыши входит в слой
    [ onmouseout="обработчик" ]  // Вызывается, когда курсор мыши выходит из слоя
    [ onfocus="обработчик" ]     // Вызывается, когда слой получает фокус
    [ onblur="обработчик" ]      // Вызывается, когда слой теряет фокус
    [ onload="обработчик" ]      // Вызывается, когда содержимое слоя загружено
>
```

Описание

Объект Layer – это технология Netscape 4, предназначенная для поддержки динамически позиционированных элементов HTML. Однако обратите внимание, что объект Layer не был стандартизован и в Netscape 6 не поддерживается. Объект Layer может быть создан одним из трех способов: с помощью тега `<layer>`, с помощью конструктора `Layer()` или, наиболее переносимым образом, с помощью атрибутов стилей CSS в элементах HTML, как объясняется в главе 18.

См. также

Window; главу 18

Layer.captureEvents()

См. Window.captureEvents()

Layer.handleEvent()

См. Window.handleEvent()

Layer.load() только Netscape 4

изменяет содержимое и ширину слоя

Синтаксис

layer.load(*src*, *width*)

Аргументы

src

Строка, задающая URL документа, который должен быть загружен в слой *layer*.

width

Целое, задающее новую ширину слоя *layer* в пикселах.

Описание

`load()` загружает в слой *`layer`* новый документ и задает ширину *`width`*, в соответствии с которой будет выполняться перенос строк этого документа.

Заметьте, однако, что `load()` не работает, пока выполняется анализ текущего документа. По этой причине нельзя вызывать `load()` в сценарии верхнего уровня. Вместо этого следует вызывать его в обработчике события или в функции, вызываемой из обработчика.

См. также

Свойство `src` объекта Layer

Layer.moveAbove() только Netscape 4

помещает один слой над другим

Синтаксис

`layer``.moveAbove(`*`target`*`)`

Аргументы

`target`

Ссылка на объект Layer, над которым должен быть помещен слой *`layer`*.

Описание

`moveAbove()` изменяет порядок расположения слоев в стеке, так что слой *`layer`* оказывается над слоем *`target`*. *`layer`* становится смежным для *`target`*, если он еще им не является. Слою *`layer`* присваивается тот же *`zIndex`*, как у слоя *`target`*, и он помещается в массив `layers[]` родительского документа после *`target`*.

Layer.moveBelow() только Netscape 4

помещает один слой над другим

Синтаксис

`layer``.moveBelow(`*`target`*`)`

Аргументы

`target`

Ссылка на объект Layer, под которым должен быть помещен слой *`layer`*.

Описание

`moveBelow()` изменяет порядок расположения слоев в стеке так, что слой *`layer`* оказывается под слоем *`target`*. *`layer`* становится смежным для *`target`*, если до этого им не является. Слою *`layer`* присваивается тот же *`zIndex`*, как у *`target`*, и он помещается в массиве `layers[]` родительского документа перед *`target`*.

Layer.moveBy() только Netscape 4

перемещает Layer на относительную позицию

Синтаксис

`layer``.moveBy(`*`dx, dy`*`)`

Аргументы

dx Количество пикселов, на которые слой должен быть перемещен вправо (может быть отрицательным).

dy Количество пикселов, на которые слой должен быть перемещен вниз (может быть отрицательным).

Описание

`moveBy()` перемещает слой *layer* на *dx* пикселов вправо и на *dy* пикселов вниз от текущей позиции.

Layer.moveTo()

только Netscape 4

перемещает слой

Синтаксис

layer`.moveTo(`*x*`, `*y*`)`

Аргументы

x – X-координата слоя.

y – Y-координата слоя.

Описание

`moveTo()` перемещает верхний правый угол слоя *layer* в точку с координатами, указанными аргументами *x* и *y*. Обратите внимание, что эти координаты выражены относительно родительского слоя или документа.

Layer.moveToAbsolute()

только Netscape 4

перемещает слой относительно страницы

Синтаксис

layer`.moveToAbsolute(`*x*`, `*y*`)`

Аргументы

x – X-координата слоя.

y – Y-координата слоя.

Описание

`moveToAbsolute()` перемещает верхний левый угол слоя *layer* в точку с координатами, заданными аргументами *x* и *y*. Важно, что координаты задаются относительно страницы или документа верхнего уровня, а не относительно какого-либо родительского слоя.

Layer.offset()

только Netscape 4; признан устаревшим

перемещает слой на относительную позицию

Синтаксис

layer`.offset(`*dx*`, `*dy*`)`

Аргументы

dx Количество пикселов, на которые слой должен быть перемещен вправо (может быть отрицательным).

dy Количество пикселов, на которые слой должен быть перемещен вниз (может быть отрицательным).

Описание

`offset()` перемещает слой относительно его текущей позиции. Метод признан устаревшим в пользу `moveBy()`.

Layer.releaseEvents()

См. Window.releaseEvents()

Layer.resizeBy() только Netscape 4

изменяет размер слоя на относительное значение

Синтаксис

layer.resizeBy(*dw*, *dh*)

Аргументы

dw Количество пикселов, на которое должна быть изменена ширина слоя (может быть отрицательным).

dh Количество пикселов, на которое должна быть изменена высота слоя (может быть отрицательным).

Описание

`resizeBy()` изменяет размер слоя *layer*, изменяя его свойства `clip.width` и `clip.height` на *dw* и *dh*. Это не вызывает переформатирование содержимого слоя, поэтому уменьшение размера слоя может привести к потере части его содержимого.

Layer.resizeTo() только Netscape 4

изменяет размер слоя

Синтаксис

layer.resizeTo(*ширина*, *высота*)

Аргументы

ширина

Желаемое значение ширины слоя.

высота

Желаемое значение высоты слоя.

Описание

`resizeTo()` изменяет размер слоя *layer* путем установки значений свойств `clip.width` и `clip.height` равными *ширине* и *высоте*. Это не приводит к переформатированию содержимого слоя, поэтому уменьшение размера слоя может привести к потере части его содержимого.

Layer.routeEvent()

См. Window.routeEvent()

Link

JavaScript 1.0; расширен в JavaScript 1.1

гиперссылка **наследуется от HTMLElement**

Синтаксис

document.links[]
document.links.length

Свойства

Link наследует свойства от HTMLElement и определяет следующие свойства. Многие из свойств представляют фрагменты URL. Для каждого из этих свойств дается пример фрагмента следующего (фиктивного) URL:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

`hash`

Доступное для чтения и записи свойство, задающее якорную часть URL ссылки, включая ведущий символ «решетка» (#). Например: «#result». Эта якорная часть URL ссылается на именованную позицию внутри документа, на который ссылается объект Link. В HTML-файлах позиции обозначаются якорными элементами, созданными с помощью тега `<a name="имя_якоря">`.

`host`

Доступное для чтения и записи строковое свойство, которое задает часть URL ссылки, содержащую имя хоста и порт. Например, «www.oreilly.com:1234».

`hostname`

Строковое свойство, доступное для чтения и записи, задающее имя хоста в URL ссылки. Например, «www.oreilly.com».

`href`

Доступное для чтения и записи строковое свойство, задающее полный текст URL ссылки, в отличие от других URL-свойств объекта Link, задающих только части URL.

`pathname`

Доступное для чтения и записи строковое свойство, задающее имя пути в URL ссылки. Например, «/catalog/search.html».

`port`

Доступное для чтения и записи строковое (не числовое) свойство, задающее порт в URL ссылки. Например, «1234».

`protocol`

Доступное для чтения и записи строковое свойство, задающее протокол в URL ссылки, включая замыкающее двоеточие. Например, «http:».

`search`

Доступное для чтения и записи строковое свойство, задающее часть URL ссылки, которая содержит строку запроса, включая ведущий вопросительный знак. Например, «?q=JavaScript&m=10».

target

Доступное для чтения и записи строковое свойство, задающее имя объекта Window (то есть фрейма или окна броузера верхнего уровня), где должен быть показан документ, на который ссылается объект Link. Подробности см. на справочной странице «Link.target».

text [Netscape 4]

Задает обычный текст, если он есть, между тегами <a> и ссылки. Обратите внимание, что это свойство корректно работает, только если между тегами <a> и нет промежуточных тегов HTML. Если есть другие теги HTML, свойство text может содержать только часть текста ссылки. HTMLElement.innerText предоставляет эквивалент в IE 4 для этого свойства, специфического для Netscape.

Методы

Link наследует методы HTMLElement.

Обработчик событий

Link наследует обработчики событий HTMLElement и определяет для следующих из них особое поведение:

onclick

Вызывается, когда пользователь щелкает по ссылке. В JavaScript 1.1 этот обработчик события может предотвратить открытие ссылки, вернув false. На платформах Windows в Netscape 3 этот обработчик события не работает для ссылок, созданных с помощью тега <area>.

onmouseout

Вызывается, когда пользователь уводит курсор мыши от ссылки. Доступен в JavaScript 1.1 и в более поздних версиях.

onmouseover

Вызывается, когда пользователь перемещает курсор мыши над ссылкой. Здесь может быть установлено свойство status окна. Может возвращать true, сообщая броузеру о том, что URL ссылки не должен быть показан.

Синтаксис HTML

Объект Link создается с помощью стандартных тегов <a> и . Для всех объектов Link атрибут href обязателен. Если также указан атрибут name, создается еще и объект Anchor:

```
<a href="url"                      // Цель ссылки
    [ name="якорь" ]               // Создавать объект Anchor
    [ target="имя_окна" ]          // Где должен быть показан новый документ
    [ onclick="обработчик" ]       // Вызывается при щелчке по ссылке
    [ onmouseover="обработчик" ]   // Вызывается, когда курсор мыши находится над ссылкой
    [ onmouseout="обработчик" ]    // Вызывается, когда курсор мыши выходит из ссылки
>
текст или изображение ссылки       // Видимая часть ссылки
</a>
```

В JavaScript 1.1 и более поздних версиях объект Link также создается каждым тегом <area> в клиентской карте изображения. Это также стандартный код HTML:

```
<map name="имя_карты">
    <area shape="форма_области"
        coords="координаты"
        href="url"                    // Цель ссылки
        [ target="имя_окна" ]         // Где должен быть показан новый документ
        [ onclick="обработчик" ]      // Вызывается при щелчке по области
        [ onmouseover="обработчик" ] // Вызывается, когда курсор мыши находится над областью
        [ onmouseout="обработчик" ]  // Вызывается, когда курсор мыши уходит из области
>
    ...
</map>
```

Описание

Объект Link представляет гиперссылку или активную область клиентской карты изображения в HTML-документе. Все ссылки, созданные с помощью тегов `<a>` и `<area>`, представляются объектами Link и сохраняются в массиве `links[]` объекта Document. Обратите внимание, что теги `<a>` и `<area>` хранятся в одном массиве – между ними не делается различия.

Целью гиперссылки является, конечно, URL, и многие свойства объекта Link задают содержимое этого URL. Объект Link похож на объект Location, который также имеет полный набор URL-свойств. В случае объекта Location эти свойства описывают URL текущего отображаемого документа.

В дополнение к этим свойствам объект Link имеет три обработчика событий. Обработчики `onmouseover()`, `onclick()` и `onmouseout()` определяют код, который должен исполняться, когда курсор мыши проходит над гиперссылкой, когда выполняется щелчок по ней и когда курсор мыши выходит из экранной области ссылки.

См. также

Anchor, Location; HTMLAnchorElement в «Справочнике по W3C DOM»

Link.onclick

JavaScript 1.0; расширен в JavaScript 1.1

обработчик, вызываемый при щелчке по ссылке

Синтаксис

```
<a ... onclick="обработчик" ... >
<area ... onclick="обработчик" ... >
link.onclick
```

Описание

Свойство `onclick` объекта Link определяет функцию обработки события, которая вызывается, когда пользователь щелкает по ссылке. Начальное значение этого свойства – функция, содержащая инструкции JavaScript, указанные в атрибуте `onclick` определяющего объект Link тега `<a>` или `<area>`. Если функция обработки события определяется подобным образом с помощью атрибута HTML, то она исполняется в области действия элемента *link*, а не в области действия родительского окна.

Обработчик события `onclick` вызывается перед тем, как броузер откроет выбранную гиперссылку. Это позволяет динамически устанавливать `href`, `target` и другие свойства ссылки (используя ключевое слово `this` для обращения к текущей ссылке). Из этого обработчика события можно также вызывать методы `Window.alert()`, `Window.confirm()` и `Window.prompt()`.

В JavaScript 1.1 можно предотвратить открытие броузером ссылки, вернув значение false. Если вернуть true, или любое другое значение, или не возвращать ничего, то броузер откроет ссылку сразу по завершении обработчика onclick. Можно предотвратить открытие броузером ссылки, если, например, вызвать метод Window.confirm() и на вопрос о том, действительно ли он хочет открыть ссылку, пользователь нажмет кнопку Cancel. Как правило, если требуется ссылка, выполняющая некоторое действие, а не приводящая к отображению нового URL, следует предпочесть обработчик события onclick объекта Button обработчику onclick объекта Link.

Обратите внимание, что обработчик события onclick возвращает false, сообщая броузеру, что тот не должен выполнять действие по умолчанию (открывать ссылку), а обработчик события onmouseover должен вернуть true, чтобы сообщить броузеру о том, что тот не должен выполнять свое действие по умолчанию (отображать URL ссылки). Эта несовместимость имеет исторические причины. Стандарт для обработчиков событий объекта Form и элементов формы состоит в возвращении false для предотвращения выполнения броузером действия по умолчанию.

В событийной модели Netscape 4 объект Event передается обработчику события onclick в качестве аргумента. В событийной модели IE аргумент не передается, но подходящий объект Event доступен как свойство event объекта Window, содержащего гиперссылку.

Ошибки

В Netscape 3 на платформах Windows обработчик события onclick тега <area> не работает. Обходное решение состоит в том, чтобы установить значение атрибута href тега <area> равным URL javascript:.

См. также

Главу 18; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Link.onmouseout

JavaScript 1.1

обработчик, вызываемый при выходе курсора мыши из ссылки

Синтаксис

```
<a ... onmouseout="обработчик" ... >
<area ... onmouseout="обработчик" ... >
link.onmouseout
```

Описание

Свойство onmouseout объекта Link определят функцию обработки события, которая вызывается, когда пользователь уводит курсор мыши от гиперссылки. Начальное значение этого свойства – функция, содержащая инструкции JavaScript, указанные в атрибуте onmouseout определяющего объект Link тега <a> или <area>. Если функция обработки события определяется подобным образом с помощью атрибута HTML, то она исполняется в области действия элемента *link*, а не в области действия родительского окна.

В событийной модели Netscape 4 объект Event передается обработчику события onmouseout в качестве аргумента. В событийной модели IE аргумент не передается, но подходящий объект Event доступен как свойство event объекта Window, содержащего гиперссылку.

См. также

Главу 18; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Link.onmouseover

JavaScript 1.0

обработчик, вызываемый при проходе курсора мыши над ссылкой

Синтаксис

```
<a ... onmouseover="обработчик" ... >
<area ... onmouseover="обработчик" ... >
link.onmouseover
```

Описание

Свойство `onmouseover` объекта Link задает функцию обработки события, вызываемую, когда пользователь перемещает курсор мыши над гиперссылкой. Начальное значение этого свойства – функция, содержащая инструкции JavaScript, указанные в атрибуте `onmouseover` определяющего объект Link тега `<a>` или `<area>`. Если функция обработки события определяется подобным образом с помощью атрибута HTML, то она исполняется в области действия элемента *link*, а не в области действия родительского окна.

По умолчанию, когда курсор мыши проходит над ссылкой, в строке состояния броузер отображает URL, на который ссылается гиперссылка. Перед отображением URL вызывается обработчик события `onmouseover`. Поэтому функция обработки события, возвращающая `true`, может вывести собственное сообщение в строку состояния, присвоив любое желаемое значение свойству `Window.status`.

Обратите внимание – этот обработчик события сообщает броузеру о том, что он не должен выполнять действие по умолчанию (отображать URL ссылки), возвращая `true`, а обработчик события `onclick` объекта Link с той же целью должен возвращать `false`, но в этом случае броузер не должен открывать ссылку. Эта несовместимость имеет исторические причины. Для обработчиков событий объекта Form и элементов формы определено стандартное поведение – они предотвращают выполнение броузером действия по умолчанию, возвращая `false`.

В событийной модели Netscape 4 объект Event передается обработчику события `onmouseover` в качестве аргумента. В событийной модели IE аргумент не передается, но подходящий объект Event доступен как свойство `event` объекта Window, содержащего гиперссылку.

См. также

Главу 18; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Link.target

JavaScript 1.0

целевое окно гиперссылки

Синтаксис

```
link.target
```

Описание

`target` – это доступное для чтения и записи строковое свойство объекта Link. Оно задает имя фрейма или окна, где должен отображаться URL, на который ссылается объект Link. Исходное значение этого свойства задается атрибутом `target` тега `<a>`,

создавшего объект Link. Если этот атрибут не установлен, по умолчанию выбирается окно, содержащее ссылку, поэтому при открытии гиперссылки документ, содержащий ссылку, очищается.

Обратите внимание, что значение *target* – это *имя* фрейма или окна, а не реальная JavaScript-ссылка на сам фрейм или окно. Имя фрейма задается атрибутом `name` тега `<frame>`. Имя окна определяется при создании окна с помощью метода `Window.open()`. Если `target` задает имя несуществующего окна, броузер автоматически открывает новое окно для отображения URL, и все последующие ссылки на то же имя *target* используют созданное окно.

Поддерживается четыре специальных имени цели. Цель с именем `"_blank"` означает, что должно быть создано новое пустое окно, в котором и должен отображаться URL. Цель `"_ self"` – это цель по умолчанию, и в этом случае новый URL должен отображаться в том же фрейме или окне, в котором находится ссылка. Цель `"_parent"` означает, что результат должен отображаться в родительском фрейме фрейма, содержащего ссылку. И наконец, если цель принимает значение `"_top"`, то новый URL должен отображаться во фрейме самого верхнего уровня. Другими словами, все фреймы должны быть удалены и новый URL должен занимать все окно броузера.

См. также

Form.target

Location

JavaScript 1.0; расширен в JavaScript 1.1

обозначает адрес в броузере и управляет им

Синтаксис

`location`
`window``.location`

Свойства

Свойства объекта Location ссылаются на различные фрагменты URL текущего документа. В каждом из следующих описаний свойств дается пример фрагмента следующего (фиктивного) URL:

```
http://www.oreilly.com:1234/catalog/search.html?q=JavaScript&m=10#results
```

`hash`

Доступное для чтения и записи свойство, задающее якорную часть URL, включая ведущий символ «решетка» (#). Например: «#result». Эта часть URL документа задает имя якорного элемента внутри документа.

`host`

Доступное для чтения и записи строковое свойство, которое задает часть URL, содержащую имя хоста и порт. Например, «www.oreilly.com:1234».

`hostname`

Строковое свойство, доступное для чтения и записи, задающее имя хоста в URL. Например, «www.oreilly.com».

`href`

Доступное для чтения и для записи строковое свойство, задающее полный текст URL документа, в отличие от других свойств объекта Location, которые определя-

ют только части URL. Присваивание этому свойству нового URL приводит к тому, что броузер читает и отображает содержимое нового URL.

`pathname`

Строковое свойство, доступное для чтения и записи, задающее путь в URL. Например, «/catalog/search.html».

`port`

Доступное для чтения и записи строковое (не числовое) свойство, задающее порт в URL ссылки. Например, «1234».

`protocol`

Доступное для чтения и записи строковое свойство, задающее протокол в URL, включая замыкающее двоеточие. Например, «http:».

`search`

Доступное для чтения и записи строковое свойство, задающее часть URL, которая содержит строку запроса, включая ведущий вопросительный знак. Например, «?q=JavaScript&m=10».

Методы

`reload()`

Повторно загружает текущий документ из кэша или с сервера. Этот метод был добавлен в JavaScript 1.1.

`replace()`

Заменяет текущее окно новым, при этом новая запись в истории сессии броузера не генерируется. Этот метод был добавлен в JavaScript 1.1.

Описание

Объект Location хранится в свойстве `location` объекта Window и представляет веб-адрес («местоположение») документа, отображаемого в данный момент в этом окне. Свойство `href` содержит полный URL этого документа, а каждое из оставшихся свойств объекта Location описывает фрагмент этого URL. Эти свойства очень похожи на URL-свойства объекта Link.

Если объект Link представляет гиперссылку в документе, то объект Location представляет URL (адрес), отображаемый в данный момент в броузере. Но объект Location еще и *управляет* адресом, отображаемым в броузере. Если присвоить строку, содержащую URL, объекту Location или его свойству `href`, то веб-броузер загрузит указанный URL и отобразит документ, на который он ссылается.

Замена текущего URL совершенно новым адресом осуществляется не только путем присваивания значения `location` или `location.href`, но и путем изменения части текущего URL (присваивания строковых значений другим свойствам объекта Location). Таким образом создается новый URL с одним измененным фрагментом, который загружается и отображается броузером. Так, установив свойство `hash` объекта Location, можно заставить броузер переместиться к именованному месту в текущем документе. Аналогично, установив свойство `search`, можно заставить броузер заново загрузить текущий URL, дополненный новой строкой запроса. Если URL ссылается на серверную программу, документ, полученный от новой строки запроса, может существенно отличаться от первоначального документа.

В дополнение к URL-свойствам объект Location также определяет два метода. Метод `reload()` повторно загружает текущий документ, а метод `replace()` загружает новый

документ без создания для него новой записи в истории – новый документ заменяет текущий в списке истории броузера.

См. также

Link, свойство `location` объекта Window

Location.reload() JavaScript 1.1

заново загружает текущий документ

Синтаксис

location.reload()
location.reload(*force*)

Аргументы

force

Логический аргумент, определяющий, должен ли документ быть загружен заново, даже если сервер сообщает о том, что документ не был изменен с момента его последней загрузки. Если этот аргумент опущен или равен `false`, метод загружает страницу целиком, только если она был изменена с момента последней загрузки.

Описание

Метод `reload()` объекта Location заново загружает документ, отображаемый в данный момент в окне, к которому относится объект Location. При вызове без аргументов или с аргументом `false` он определяет, был ли документ изменен на веб-сервере, по HTTP-заголовку `If-Modified-Since`. Если документ был изменен, метод `reload` заново загружает документ с сервера, а если нет, загружает его из кэша. Это то же самое действие, которое выполняется, когда пользователь щелкает по кнопке Reload броузера.

Когда метод `reload()` вызывается с аргументом, равным `true`, он всегда выполняется в обход кэша и загружает документ с сервера независимо от времени последней модификации документа. Это действие идентично тому, которое выполняется, когда пользователь щелкает по кнопке Reload броузера при нажатой клавише Shift.

Location.replace() JavaScript 1.1

заменяет один изображаемый документ другим

Синтаксис

location.replace(*url*)

Аргументы

url

Строка, задающая URL нового документа, который должен заменить текущий.

Описание

Метод `replace()` объекта Location загружает и отображает новый документ. Важное отличие этого способа загрузки документа от простого присваивания значения *location* или *location*.`href` состоит в том, что метод `replace()` не генерирует новую запись в объекте History. В последнем случае новый URL заменяет текущую запись в объекте History. После вызова `replace()` кнопка Back броузера возвращает нас не к предыдущему URL, а к предпоследнему.

Применение

При работе с несколькими фреймами и/или с документами, сгенерированными JavaScript, может образоваться довольно много временных документов. Если этих документов слишком много, обратный обход вашего веб-сайта с помощью кнопки Back может быть не очень приятным занятием. Однако применение метода `replace()` для загрузки этих документов помогает избежать этой неприятности.

См. также

History

MimeType

Netscape 3

представляет MIME-тип данных

Синтаксис

```
navigator.mimeTypes[i]
navigator.mimeTypes["тип"]
navigator.mimeTypes.length
```

Свойства

`description`

Строка, доступная только для чтения, которая предоставляет понятное человеку описание (по-английски) типа данных, характеризуемого MimeType. Это описание более подробно и понятно, чем свойство `name`.

`enabledPlugin`

Доступная только для чтения ссылка на объект Plugin, представляющий установленный и включенный плагин, управляющий указанным MIME-типом. Если такого плагина нет, то значение этого свойства равно `null`.

Массив `navigator.mimeType[]` сообщает о том, поддерживается ли данный MIME-тип броузером. А свойство `enabledPlugin` объекта MimeType сообщает, поддерживается ли определенный совместимый тип с помощью подключаемого модуля (MIME-типы могут также поддерживаться вспомогательными приложениями, а не непосредственно броузером). Если MIME-тип поддерживается подключаемым модулем, данные этого типа могут встраиваться в веб-страницу с помощью тега `<embed>`.

`suffixes`

Доступная только для чтения строка, содержащая список суффиксов имен (через запятую, не включая символ «.»), обычно используемых для файлов указанного MIME-типа. Например, суффиксы для MIME-типа `text/html` — «html, htm».

`type`

Доступная только для чтения строка, задающая имя MIME-типа. Это уникальная строка, такая как «text/html» или «image/jpeg», отличающая данный MIME-тип от всех остальных. Она описывает общий тип данных и используемый формат данных. Значение свойства `type` может также выступать в качестве индекса для доступа к элементам в массиве `navigator.mimeTypes[]`.

Описание

Объект MimeType представляет MIME-тип (то есть формат данных), поддерживаемый в Netscape. Формат может поддерживаться для встраиваемых данных либо непосредственно броузером, либо через внешнее вспомогательное приложение или плагин.

Применение

Массив navigator.mimeTypes[] может индексироваться по номеру или по имени требуемого MIME-типа (являющегося значением свойства type). Для того чтобы проверить, какие MIME-типы поддерживаются в Netscape, можно выполнить числовой цикл по всем элементам массива. Если же надо просто проверить, поддерживается ли определенный тип, можно написать следующий код:

```
var show_movie = (navigator.mimeTypes["video/mpeg"] != null);
```

См. также

Navigator, Plugin

Navigator

JavaScript 1.0; расширен в JavaScript 1.1 и 1.2

информация об используемом броузере

Синтаксис

navigator

Свойства

appCodeName

Доступная для чтения и записи строка, задающая кодовое имя броузера. Во всех броузерах Netscape она равна «Mozilla». Для совместимости в броузерах от Microsoft она также равна «Mozilla» .

appName

Доступное только для чтения свойство, задающее имя броузера. Для Netscape значение этого свойства равно «Netscape». В Internet Explorer оно равно «Microsoft Internet Explorer».

appVersion

Доступная только для чтения строка, задающая информацию о версии и платформе броузера. Первая часть этой строки представляет собой номер версии. Чтобы получить только старший номер версии, передайте строку в метод parseInt(), а чтобы получить старший и младший номера версии – в метод parseFloat(). Остальная часть строкового значения этого свойства предоставляет другие сведения о версии броузера, включая операционную систему, на которой он работает. Однако, к сожалению, формат этой информации сильно отличается в разных броузерах.

cookieEnabled *[IE 4, Netscape6]*

Доступное только для чтения логическое значение, равное true, если в броузере включены cookie, и false, если они отключены.

language *[Netscape 4]*

Доступная только для чтения строка, задающая язык по умолчанию данной версии броузера. Это свойств равно либо стандартному двухбуквенному коду языка, например «en» для английского языка или «fr» для французского, либо пятибуквенной строке, обозначающей язык и его региональный вариант, например «fr_CA» для французского языка, на котором говорят в Канаде. Обратите внимание, что IE 4 предоставляет два различных свойства, имеющих отношение к языку.

`mimeTypes[]` *[Netscape 3]*

Массив объектов MimeType, каждый из которых представляет один из MIME-типов (например, «text/html» и «image/gif»), поддерживаемый броузером. Массив `mimeTypes[]` определен в IE 4, но всегда пуст, так как IE 4 не поддерживает объект MimeType.

`platform` *[JavaScript 1.2]*

Доступная только для чтения строка, задающая операционную систему и/или аппаратную платформу, на которой запущен броузер. Стандартный набор значений для этого свойства не определен, но вот некоторые типичные значения: «Win32», «MacPPC» и «Linux i586».

`plugins[]` *[Netscape 3]*

Массив объектов Plugin, каждый из которых представляет один плагин, установленный в броузере. Объект Plugin предоставляет информацию о подключаемом модуле, в том числе список поддерживаемых им MIME-типов. Подключаемый модуль (plugin) – это принятое в Netscape название программного пакета, вызываемого броузером для отображения определенных типов данных в окне броузера.

Массив `plugins[]` определен в IE 4, но всегда пуст, так как IE 4 не поддерживает подключаемые модули и объект Plugin.

`systemLanguage` *[IE 4]*

Доступная только для чтения строка, задающая язык по умолчанию в операционной системе с помощью тех же стандартных кодов, которые используются в специфическом для Netscape свойстве `language`.

`userAgent`

Доступная только для чтения строка, задающая значение, которое броузер подставляет в заголовок user-agent в HTTP-запросах. Обычно оно равно значению свойства `navigator.appCodeName`, за которым следует косая черта и значение свойства `navigator.appVersion`. Например:

```
Mozilla/4.0 (compatible; MSIE 4.01; Windows 95)
```

`userLanguage` *[IE 4]*

Строка, доступная только для чтения, задающая предпочтительный для пользователя язык при помощи тех же стандартных кодов, которые применяются в специфическом для Netscape свойстве `language`.

Функции

`navigator.javaEnabled()`

Проверяет, поддерживается ли Java и включена ли эта поддержка в текущем броузере. Добавлена в JavaScript 1.1.

`navigator.plugins.refresh()`

Проверяет на наличие только что установленных подключаемых модулей, заносит их в массив `plugins[]` и (необязательно) заново загружает документы с помощью этих модулей. Добавлена в Netscape 3.

Описание

Объект Navigator содержит свойства, описывающие используемый веб-броузер. Посредством этих свойств можно выполнить специфическую для платформы настройку сценария. Имя этого объекта, очевидно, ссылается на броузер Netscape Navigator, но другие броузеры, поддерживающие JavaScript, также поддерживают этот объект.

Имеется только один экземпляр объекта Navigator, к которому можно обращаться через свойство navigator любого объекта Window. Поскольку существует неявная ссылка на объект Window, то к объекту Navigator всегда можно обратиться просто как к navigator.

См. также

MimeType, Plugin

Navigator.javaEnabled() JavaScript 1.1

проверяет, включена ли поддержка Java

Синтаксис

navigator.javaEnabled()

Возвращаемое значение

true, если поддержка Java включена в текущем броузере; false, если нет.

Описание

Метод navigator.javaEnabled() позволяет проверить, поддерживает ли текущий броузер Java и, следовательно, поддерживает ли он апплеты.

Navigator.plugins.refresh() Netscape 3

делает доступными только что установленные подключаемые модули

Синтаксис

navigator.plugins.refresh([*reload*])

Аргументы

reload

Необязательный логический аргумент, который, будучи равным true, означает, что метод refresh() должен заново загрузить любые страницы, содержащие теги <embed> и использующие подключаемые модули. Если опущен, то по умолчанию имеет значение false.

Описание

Вызов метода refresh() заставляет Netscape проверить, были ли установлены какие-либо новые подключаемые модули. Если это так, массив plugins[] обновляется и в него помещается информация о только что установленных подключаемых модулях. Если аргумент *reload* задан и равен true, Netscape также загружает заново все отображаемые в данный момент документы, содержащие теги <embed> и использующие подключаемые модули.

Обратите внимание на необычный синтаксис этого метода. refresh() — это метод массива plugins[], а не объекта Navigator. Однако в большинстве случаев проще рассматривать его как метод объекта Navigator, поэтому здесь он объединен с методами и свойствами этого объекта.

Option

JavaScript 1.0; расширен в JavaScript 1.1

опция в теге Select **наследуется от HTMLElement**

Синтаксис

select.options[*i*]

Конструктор

В JavaScript 1.1 объекты Option могут создаваться динамически с помощью конструктора Option():

```
new Option(text, value, defaultSelected, selected)
```

Аргументы

text

Необязательный строковый аргумент, задающий значение свойства text объекта Option.

value

Необязательный строковый аргумент, задающий значение свойства value объекта Option.

defaultSelected

Необязательный логический аргумент, задающий значение свойства defaultSelected объекта Option.

selected

Необязательный логический аргумент, задающий значение свойства selected объекта Option.

Свойства

Option наследует свойства HTMLElement и определяет следующие:

defaultSelected

Логическое значение, указывающее, была ли выбрана опция изначально, когда создавался содержащий ее объект Select. Это значение применяется для восстановления объекта Select к его начальному состоянию при сбросе формы. Начальное значение этого свойства задается атрибутом selected тега <option>.

index

Доступное только для чтения целое значение, задающее позицию или индекс опции в массиве options[] объекта Select, в котором она содержится. Первый объект Option в массиве имеет свойство index, равное 0, второй, – равное 1, и т. д.

selected

Доступное для чтения и записи логическое значение, указывающее, выбрана ли опция в данный момент. Посредством этого свойства можно проверить, выбрана ли данная опция, а также выбрать (путем установки ее значения в true) данную опцию или отменить ее выбор (путем установки его значения false). Обратите внимание, что при таком выборе или отмене выбора опции обработчик события Select.onchange() не вызывается.

text

Строка, задающая текст для опции, который видит пользователь. В качестве начального значения этого свойства может выступать любой текст (без тегов HTML),

расположенный после тега <option> и перед любым из следующих тегов <option>, </option> или </select>.

В JavaScript 1.0 свойство text доступно только для чтения. В JavaScript 1.1 оно доступно и для чтения и для записи. Установив новое значение для этого свойства, можно изменить отображаемый в объекте Select текст опции. Обратите внимание, что если это свойство будет установлено в броузере, который не умеет переформатировать содержимое документа, то необходимо сделать так, чтобы изменение метки опции не приводило к увеличению ширины объекта Select. А если объект становится шире, то надо сделать так, чтобы при этом не пропала никакая информация, расположенная справа от него.

value

Доступная для чтения и для записи строка, задающая текст, который передается веб-серверу, если при передаче данных формы выбрана данная опция. Начальное значение свойства value задается атрибутом value тега <option>. Если форма предназначена для отправки на сервер (в противоположность использованию в JavaScript на стороне клиента), все объекты Option внутри объекта Select должны иметь различные значения свойства value.

Синтаксис HTML

Объект Option создается с помощью тега <option> внутри тега <select>, находящегося внутри <form>. В теге <select> обычно присутствуют несколько тегов <option>:

```
<form ...>
  <select  ...>
    <option
        [ value="значение" ]   // Значение, возвращаемое при передаче формы
        [ selected ] >         // Указывает, выбрана ли эта опция первоначально
    текстовая_метка            // Текст, отображаемый для этой опции
    [ </option> ]
        ...
  </select>
        ...
</form>
```

Описание

Объект Option описывает одиночную опцию, отображаемую внутри объекта Select. Свойства этого объекта определяют, выбрана ли она по умолчанию, выбрана ли она в данный момент, задают позицию, которую она занимает в массиве options[] содержащего ее объекта Select, отображаемый ею текст и значение, которое она передает на сервер при передаче данных родительской формы.

Обратите внимание, что хотя текст, отображаемый этой опцией, задается вне тега <option>, он должен быть обычным, неформатированным текстом, без каких-либо тегов HTML, чтобы он мог нормально отображаться в списках и выпадающих меню, которые не поддерживают средства форматирования HTML.

В JavaScript 1.1 можно динамически создавать новые объекты Option для отображения в объекте Select с помощью конструктора Option(). Создав новый объект Option, можно добавить его к списку опций в объекте Select, присвоив *s*:

```
s.options[options.length]
```

Подробности см. на справочной странице «Select.options[]».

См. также

Select, Select.options[]; HTMLOptionElement и HTMLSelectElement в «Справочнике по W3C DOM»

Password

JavaScript 1.0; расширен в JavaScript 1.1

текстовое поле ввода для секретных данных **наследуется от Input, HTMLElement**

Синтаксис

form.name
form.elements[*i*]

Свойства

Объект Password наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

value

Доступная для чтения и записи строка, задающая пароль, введенный пользователем. Это значение, которое передается по сети при передаче данных формы. Начальное значение этого свойства задается атрибутом value элемента <input>, определяющего объект Password. Обратите внимание, что пароль должен быть засекречен, и поэтому значение свойства value может быть защищено ограничениями политики безопасности. В некоторых броузерах строка, возвращаемая при запросе этого свойства, не совпадает с текстом, введенным пользователем, и установка этого свойства может не влиять на отображаемое значение или на значение, пересылаемое при передаче данных формы.

Методы

Объект Password наследует методы от Input и HTMLElement.

Обработчик событий

Объект Password наследует обработчики событий от Input и HTMLElement.

Синтаксис HTML

Элемент Password создается с помощью стандартного тега HTML <input>:

```
<form>
    ...
  <input
    type="password"              // Указывает, что это элемент Password
    [ name="имя" ]               // Имя, по которому позднее можно сослаться на этот элемент
                                 // Задает значение свойства name
    [ value="по_умолчанию" ]     // Значение, передаваемое при отправке формы
    [ size="целое" ]             // Ширина элемента в символах
  >
    ...
</form>
```

Описание

Элемент Password – это поле ввода текста, предназначенное для ввода секретных данных, таких как пароли. При наборе символов пользователем в поле появляются только символы звездочек. Это не дает возможности наблюдателю прочитать введенное значение из-за спины пользователя. В качестве дополнительной предосторож-

ности установлены ограничения на способы чтения и записи значения свойства value элемента Password из JavaScript. Дополнительную информацию см. на справочных страницах «Text» и «Input».

См. также

Input, Text; HTMLInputElement в «Справочнике по W3C DOM»

Plugin

Netscape 3

описывает установленный подключаемый модуль

Синтаксис

navigator.plugins[*i*]
navigator.plugins['*имя*']

Свойства

description

Строка, доступная только для чтения, которая содержит понятное человеку описание указанного подключаемого модуля. Текст этого описания предоставляется создателями сценария и может содержать информацию о разработчике и версии, а также краткое описание функционирования модуля.

filename

Доступная только для чтения строка, задающая имя файла на диске, который содержит собственно программу подключаемого модуля. Это имя может отличаться от платформы к платформе. Для идентификации модуля более полезным, чем filename, является свойство name.

length

Каждый объект Plugin содержит элементы массива MimeType, задающие форматы данных, которые поддерживаются подключаемым модулем. Как и во всех массивах, свойство length задает количество элементов в массиве.

name

Свойство name объекта Plugin – это доступная только для чтения строка, задающая имя подключаемого модуля. Каждый модуль должен иметь имя, уникально идентифицирующее его. Имя может выступать в качестве индекса для обращения к массиву navigator.plugins[]. Используя этот факт, нетрудно определить, установлен ли указанный подключаемый модуль в текущем броузере:

```
var sw_installed = (navigator.plugins["Shockwave"] != null);
```

Элементы

Элементы массива объекта Plugin – это объекты MimeType, задающие форматы данных, которые поддерживаются подключаемым модулем.

Описание

Подключаемый модуль (*plugin*) – это программный модуль, который может вызываться Netscape для отображения специализированных типов данных, встраиваемых в окно броузера. В Netscape 3 подключаемые модули представлены объектом Plugin. Этот объект несколько необычен тем, что он имеет как обычные свойства объекта, так и элементы массива. Свойства объекта Plugin предоставляют различную информа-

цию о подключаемом модуле, а его элементы массива – это объекты MimeType, задающие форматы встраиваемых данных, которые поддерживает этот модуль.

Объекты Plugin могут быть получены из массива plugins[] объекта Navigator. navigator.plugins[] может индексироваться численно, когда требуется выполнить цикл по всему списку установленных подключаемых модулей, чтобы найти нужный (например, поддерживающий MIME-тип данных, которые надо встроить в веб-страницу). Однако этот массив также может индексироваться по имени подключаемого модуля. Вот так можно проверить, установлен ли определенный модуль в броузере пользователя:

```
document.write( navigator.plugins("Shockwave") ?
                "<embed src="movie.dir' height=100 width=100>" :
                "У вас отсутствует подключаемый модуль Shockwave!" );
```

Имя, используемое в качестве индекса массива подобным образом, совпадает со значением свойства name объекта Plugin.

Не путайте тот факт, что объекты Plugin хранятся в массиве в объекте Navigator, с тем, что каждый объект Plugin сам является массивом объектов MimeType. Поскольку существует два массива, может получиться следующий код:

```
navigator.plugins[i][j]            // j-й MIME-тип в i-м плагине
navigator.plugins["LiveAudio"][0]  // Первый MIME-тип плагина LiveAudio
```

И наконец, обратите внимание, что хотя элементы массива в объекте Plugin задают MIME-типы, поддерживаемые этим подключаемым модулем, можно также определить, какой модуль поддерживает данный MIME-тип, с помощью свойства enabledPlugin объекта MimeType.

См. также

Navigator, MimeType

Radio

JavaScript 1.0; расширен в JavaScript 1.1

переключатель **наследуется от Input, HTMLElement**

Синтаксис

Переключатель обычно применяется в группах взаимоисключающих опций, имеющих одинаковое имя. Синтаксис обращения к одному элементу Radio в группе таков:

form.имя_переключателя[*j*]
form.имя_переключателя.length

Свойства

Объект Radio наследует свойства от Input и HTMLElement и определяет следующее:

checked

Доступное для чтения и для записи логическое значение, равное true, если переключатель выбран, и false в противном случае. Если установить checked равным true, то переключатель выбирается, а выбор всех остальных переключателей группы отменяется. Заметьте, однако, что установка свойства checked переключателя равным false не будет иметь эффекта; отменить выбор переключателя можно только посредством выбора другого переключателя группы. Также обратите внимание, что установка свойства checked не приводит к вызову обработчика onclick элемента Radio. Поэтому вызывать этот обработчик следует явно.

defaultChecked

Логическое свойство, равное true, если переключатель выбран изначально, но только если атрибут checked присутствует в теге HTML <input> этой кнопки. Если этот атрибут отсутствует, переключатель изначально не выбран, и свойство defaultChecked равно false.

value

Доступная для чтения и для записи строка, задающая текст, который передается на веб-сервер, если при передаче данных формы переключатель выбран. Начальное значение этого свойства задается атрибутом value тега <input> кнопки. Если форма предназначена для отправки на сервер (в противоположность использованию в JavaScript на стороне клиента), все кнопки в группе должны иметь разные значения value. Обратите внимание, что поле value не определяет, выбрана ли кнопка в данный момент; текущее состояние объекта Radio задается свойством checked.

Методы

Объект Radio наследует методы от Input и HTMLElement.

Обработчик событий

Radio наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующие:

onclick

Вызывается при щелчке по переключателю.

Синтаксис HTML

Элемент Radio создается с помощью стандартного тега HTML <input>. Элементы Radio создаются в группах путем задания нескольких тегов <input> с одинаковыми атрибутами name:

```
<form>
    ...
  <input
    type="radio"              // Указывает, что это переключатель

    [ name="имя" ]            // Имя, по которому можно будет ссылаться на эту кнопку,
                              // или для группировки кнопок с этим именем
                              // Задает значение свойства name
    [ value="значение" ]      // Значение, возвращаемое при выборе этой кнопки
                              // Задает значение свойства value
    [ checked ]               // Указывает, что эта кнопка изначально отмечена
                              // Задает свойство defaultChecked
    [ onclick="обработчик" ]  // Инструкции JavaScript, исполняемые при щелчке по кнопке
 >
метка                         // Текст HTML, который должен появиться рядом с кнопкой
    ...
</form>
```

Описание

Элемент Radio представляет одиночный графический элемент «переключатель» в форме HTML. *Переключатель* – это одна кнопка в группе кнопок, представляющих взаимоисключающие варианты выбора. Когда выбирается одна кнопка, выбор пре-

дыдущей кнопки отменяется. Обработчик события onclick позволяет задать код JavaScript, подлежащий исполнению при выборе кнопки.

Значение свойства checked позволяет определять состояние кнопки. Устанавливая это свойство, можно выбрать кнопку или отменить ее выбор. Обратите внимание, что установка свойства checked изменяет графический внешний вид кнопки, но не вызывает обработчик события onclick. Начальное значение свойства checked и значение defaultChecked определяются атрибутом checked. Только один элемент Radio в группе может содержать этот атрибут – он устанавливает значения свойств checked и defaultChecked равными true для этого элемента и равными false для всех остальных переключателей в группе. Если ни один из элементов не имеет атрибута checked, то по умолчанию отмечается первый элемент в группе (и в нем устанавливаются значения свойств checked и defaultChecked).

Обратите внимание, что текст, расположенный рядом с переключателем, не является частью самого элемента Radio и должен задаваться вне тега HTML <input> элемента Radio.

Элементы Radio применяются в группах взаимоисключающих опций. Взаимоисключающая группа определяется в виде набора всех элементов Radio в форме, имеющих одинаковое имя. Если общее имя группы элементов Radio равно f, то f.opts будет представлять собой массив элементов Radio, а f.opts.length – количество элементов в массиве.

Программист может установить атрибут value или свойство value элемента Radio для задания строки, передаваемой на сервер, если при передаче данных формы этот элемент будет отмечен. Все элементы Radio в группе должны иметь различные значения value, чтобы сценарий на сервере мог определить, какой из них был отмечен при передаче данных формы.

Применение

Элементы Radio предоставляют пользователю список нескольких взаимоисключающих опций. Список опций, не являющихся взаимоисключающими, создается при помощи элемента Checkbox.

См. также

Checkbox, Form, Input; HTMLInputElement в «Справочнике по W3C DOM»

Radio.onclick

JavaScript 1.0

обработчик, вызываемый при выборе переключателя

Синтаксис

```
<input type="radio" onclick="обработчик" ... >
radio.onclick
```

Описание

Свойство onclick объекта Radio ссылается на функцию обработки события, вызываемую, когда пользователь щелкает по переключателю. Подробности см. на справочной странице «HTMLElement.onclick». Заметьте, однако, что Radio.onclick поддерживается начиная с JavaScript 1.0, в отличие от обобщенного обработчика HTMLElement.onclick.

См. также

HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Reset

JavaScript 1.0; расширен в JavaScript 1.1

кнопка для сброса значений формы **наследуется от Input, HTMLElement**

Синтаксис

form.*name*
form.elements[*i*]

Свойства

Reset наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

value

Строка, задающая текст, который появляется внутри кнопки Reset. Он задается атрибутом value тега <input>, создавшего кнопку. Если атрибут value не указан, значение по умолчанию равно «Reset» (или эквивалентному значению на языке по умолчанию броузера). В броузере, не допускающем переформатирования содержимого документа, это свойство может быть доступно только для чтения.

Методы

Reset наследует методы от Input и HTMLElement.

Обработчик событий

Reset наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующие:

onclick

Вызывается при щелчке по кнопке Reset.

Синтаксис HTML

Элемент Reset создается с помощью стандартного тега HTML <input>:

```
<form>
    ...
  <input
    type="reset"                // Указывает, что это кнопка Reset
    [ value="метка" ]           // Текст, который должен появляться внутри кнопки
                                // Задает свойство value
    [ name="имя" ]              // Имя, по которому впоследствии можно ссылаться на кнопку
                                // Задает значение свойства name
    [ onclick="обработчик" ]    // Инструкции JavaScript, подлежащие исполнению
                                // при щелчке по кнопке
  >
    ...
</form>
```

Объекты Reset также могут создаваться с помощью тега HTML 4 <button>:

```
<button id="имя"
        type="reset"
        onclick="обработчик">
метка
</button>
```

Описание

Элемент Reset поддерживает те же свойства и методы, что элемент Button, но имеет более специальное назначение. Когда выполняется щелчок по элементу Reset, значения всех элементов ввода в форме, в которой он содержится, сбрасываются в их первоначальные значения по умолчанию. (Для большинства элементов это подразумевает значение, заданное в атрибуте HTML `value`.) Если начальное значение не задано, щелчок по кнопке Reset очищает в этих элементах все, что было введено пользователем.

Применение

Если для элемента Reset атрибут `value` не указан, на нем указывается метка «Reset». В некоторых формах лучше помечать эту кнопку как «Очистить форму» или «Значения по умолчанию».

В JavaScript 1.1 можно имитировать действие кнопки Reset посредством метода `reset()` объекта Form. Также в JavaScript 1.1 перед сбросом формы вызывается обработчик события `onreset` объекта Form. Этот обработчик может отменить сброс, вернув `false`.

См. также

Button, Form, HTMLElement, Input; HTMLInputElement в «Справочнике по W3C DOM»

Reset.onclick

JavaScript 1.0; расширен в JavaScript 1.1

обработчик, вызываемый при щелчке по кнопке Reset

Синтаксис

```
<input type="reset" onclick="обработчик" ... >
reset.onclick
```

Описание

Свойство `onclick` объекта Reset ссылается на функцию обработки события, которая вызывается при щелчке по кнопке Reset. Полное описание см. на справочной странице «HTMLElement.onclick». Однако обратите внимание, что `Reset.onclick` поддерживается начиная с JavaScript 1.0, в отличие от обобщенного обработчика `HTMLElement.onclick`.

Кнопка Reset имеет специальную функцию сброса всех элементов формы в их значения по умолчанию. Обработчик события `onclick` может добавить к кнопке Reset любую дополнительную функциональность. В JavaScript 1.1 обработчик `onclick` может возвращать `false` для предотвращения сброса формы объектом Reset. (Например, обработчик `onclick` может вызывать метод `confirm()`, запрашивающий подтверждение пользователя на сброс и возвращающий `false`, если подтверждения не было.)

См. также

Form.onreset, Form.reset(), HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Screen

JavaScript 1.2

предоставляет информацию о дисплее

Синтаксис

screen

Свойства

availHeight

Указывает доступную высоту экрана (в пикселах), на котором отображается веб-броузер. В таких операционных системах, как Windows, эта доступная высота не включает пространство, занятое полупостоянными элементами, такими как панель задач в нижней части экрана.

availLeft *[Netscape 4]*

Задает крайнюю левую координату X, не занятую полупостоянным экранным элементом, например панелью быстрого доступа или панелью задач Windows 95.

availTop *[Netscape 4]*

Задает крайнюю верхнюю координату Y, не занятую полупостоянным экранным элементом, например панелью быстрого доступа или панелью задач Windows 95.

availWidth

Задает доступную ширину экрана (в пикселах), на котором отображается веб-броузер. В таких операционных системах, как Windows, эта доступная ширина не включает пространство, занимаемое полупостоянными элементами, такими как панели быстрого доступа к приложениям.

colorDepth

Задает логарифм по основанию 2 от числа цветов, выделенных веб-броузером и доступных для вывода изображений. Например, если броузер заранее выделил 128 цветов, screen.colorDepth будет равно 7. Если система не выделяет цветовые палитры, это значение совпадает с числом битов на пиксел в экранном изображении.

В IE 4 colorDepth задает глубину цвета на экране (в битах на пиксел), а не глубину заранее выделенной цветовой палитры. Это значение в Netscape предоставляется свойством screen.pixelDepth.

height

Задает общую высоту экрана (в пикселах), на котором отображается веб-броузер. См. также availHeight.

pixelDepth *[Netscape 4]*

Задает глубину цвета (в битах на пиксел) экранного отображения веб-броузера. Сравните с colorDepth.

width

Задает общую ширину экрана (в пикселах), на котором отображается веб-броузер. См. также availWidth.

Описание

Свойство screen любого объекта Window ссылается на объект Screen. Статические свойства этого глобального объекта содержат информацию об экране, на котором отображается броузер. Программы JavaScript могут руководствоваться этой инфор-

мацией для оптимизации вывода в соответствии с возможностями дисплея пользователя. Например, программа может выбирать между большими и маленькими изображениями в зависимости от размера экрана и между изображениями с 16-битовым и 8-битовым цветом в зависимости от глубины цвета. Программа JavaScript может также использовать информацию о размере экрана для создания новых окон в центре экрана.

См. также

Свойство `screen` объекта Window

Select

JavaScript 1.0; расширен в JavaScript 1.1

графический список для выбора **наследуется от Input, HTMLElement**

Синтаксис

form.имя_элемента
form.elements[*i*]

Свойства

Объект Select наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

`length`

Доступное только для чтения целое, задающее количество элементов в массиве `options[]`. Значение этого свойства совпадает с `options.length`.

`options`

Массив объектов Option, каждый из которых описывает одну из опций, показываемых внутри элемента Select. Подробные сведения об этом массиве можно найти на справочной странице «Select.options[]», там же рассмотрены приемы изменения опций, показываемых объектом Select.

`selectedIndex`

Целое значение, задающее индекс опции, выбранной в объекте Select. Если опция не выбрана, `selectedIndex` равно –1. Если выбрана более чем одна опция, `selectedIndex` задает индекс только первой из них.

В JavaScript 1.0 `selectedIndex` доступно только для чтения. В JavaScript 1.1 это свойство доступно для чтения и для записи. Установка значения этого свойства приводит к выбору указанной опции и отменяет выбор всех остальных опций, даже если в объекте Select указан атрибут `multiple`. Если выбор реализован в виде списка (а не в виде выпадающего меню), то выбор всех опций можно отменить, установив `selectedIndex` равным 〉1. Обратите внимание, что этот способ изменения выбора не приводит к вызову обработчика события `onchange()`.

`type` *[JavaScript 1.1]*

Доступное только для чтения строковое свойство, общее для всех элементов форм; оно задает тип элемента. Объект Select необычен в том плане, что для него имеется два возможных значения свойства `type`. Если объект Select допускает только одиночный выбор (то есть если в определении объекта на HTML не указан атрибут `multiple`), свойство `type` имеет значение «select-one». Если атрибут `multiple` указан, значением атрибута `type` является «select-multiple». См. также справочную страницу «Input.type».

Методы

Select наследует методы Input и HTMLElement.

Обработчик событий

Select наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующий:

onchange

Вызывается, когда пользователь выбирает элемент списка или отменяет выбор элемента.

Синтаксис HTML

Элемент Select создается с помощью стандартного тега HTML <select>. Опции, присутствующие внутри элемента Select, создаются с помощью тега <option>:

```
<form>
    ...
<select
    name="имя"                  // Имя, идентифицирующее этот элемент; задает свойство name
    [ size="целое" ]            // Количество видимых в элементе Select опций
    [ multiple ]                // Если указано, может быть выбрано несколько опций
    [ onchange="обработчик" ]   // Вызывается при изменении выбора
>
<option value="значение1" [selected]> метка_опции1
<option value="значение2" [selected]> метка_опции2

// Другие опции
</select>
   ...
</form>
```

Описание

Элемент Select представляет графический список выбора для пользователя. Если в определении элемента на HTML присутствует атрибут multiple, пользователь может выбирать из списка любое число опций. Если этот атрибут не присутствует, пользователь может выбрать только одну опцию, и опции ведут себя как переключатели – выбор одной из них приводит к отмене предыдущего выбора.

Опции в элементе Select могут отображаться двумя различными способами. Если атрибут size имеет значение, большее 1, или если присутствует атрибут multiple, они отображаются в окне броузера в виде списка высотой size строк. Если значение size меньше, чем число опций, в списке появляется полоса прокрутки, чтобы были доступны все опции. С другой стороны, если size равно 1 и атрибут multiple не указан, текущая выбранная опция отображается в одной строке, а список всех остальных опций доступен через выпадающее меню. Первый стиль представления лучше показывает опции, но требует больше пространства в окне броузера. Второй стиль требует минимум пространства, но не показывает альтернативные опции так явно.

Самый большой интерес представляет свойство options[] элемента Select. Это массив объектов Option, описывающий варианты выбора, представленные в элементе Select. Свойство length задает длину массива (как и options.length). Подробности см. в описании объекта Option.

В JavaScript 1.1 опции, показанные в элементе Select, могут динамически изменяться. Текст, отображаемый объектом Option, можно изменить, установив значение его

свойства text. Количество опций, отображаемых в элементе Select, изменяется посредством установки свойства options.length. Кроме того, можно создавать новые опции с помощью функции-конструктора Option(). Подробности см. на справочных страницах «Select.options[]» и «Option».

Обратите внимание, что объект Select – это особый вид объекта Input, и он наследуется от Input, хотя объекты Select создаются не с помощью тегов HTML <input>.

См. также

Form, HTMLElement, Input, Option; HTMLSelectElement в «Справочнике по W3C DOM»

Select.onchange

JavaScript 1.0

обработчик, вызываемый при изменении выбора в списке

Синтаксис

```
<select ... onchange="обработчик" ... >
select.onchange
```

Описание

Свойство onchange объекта Select ссылается на функцию обработки события, вызываемую, когда пользователь выбирает опцию или отменяет ее выбор. Более подробно этот обработчик событий рассмотрен на справочной странице «Input.onchange».

См. также

Input.onchange, Option; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Select.options[]

JavaScript 1.0; расширен в JavaScript 1.1

варианты выбора в объекте Select

Синтаксис

```
select.options[i]
select.options.length
```

Описание

Свойство options[] содержит массив объектов Option, каждый из которых описывает один из вариантов выбора, показанных в объекте Select *select*. Свойство options.length задает количество элементов в массиве так же, как свойство *select*.length. Более подробные сведения содержатся в описании объекта Option.

В JavaScript 1.1 можно изменять опции, отображаемые в объекте Select, любым из следующих способов:

- Если установить options.length равным 0, все опции в объекте Select удаляются.
- Если установить options.length равным значению, меньшему текущего значения, количество опций в объекте Select уменьшается, а те опции, которые находятся в конце массива, исчезают.
- Если установить элемент массива options[] равным null, эта опция удалится из объекта Select, а элементы, расположенные в массиве над ним, переместятся

вниз, при этом их индексы изменятся таким образом, чтобы элементы заняли освободившееся место в массиве.

- Если новый объект Option создается с помощью конструктора `Option()` (см. справочную статью «Option»), то можно будет добавить эту опцию в конец списка опций в объекте Select, присвоив только что созданную опцию позиции в конце массива `options[]`. Это можно сделать при помощи выражения `options[options.length]`.

См. также

Option

Style

Internet Explorer 4, Netscape 6

атрибуты каскадной таблицы стилей

Синтаксис

htmlElement.style

Свойства

В объекте Style имеются свойства, соответствующие всем CSS-атрибутам, которые поддерживаются броузером.

Описание

Свойства объекта Style непосредственно соответствуют CSS-атрибутам, поддерживаемым в броузере. Однако для совместимости с JavaScript имена атрибутов CSS, содержащие дефис, записываются в смешанном регистре без дефисов. Поэтому, например, CSS-атрибут `color` представлен свойством `color` объекта Style, а CSS-атрибут `background-color` представлен свойством `backgroundColor` объекта Style. Дополнительную информацию о стилях элементов см. в главе 18.

См. также

HTMLElement.style; главу 18; CSSStyleDeclaration и CSS2Properties в «Справочнике по W3C DOM».

Submit

JavaScript 1.0; расширен в JavaScript 1.1

кнопка для передачи данных формы **наследуется от Input, HTMLElement**

Синтаксис

form.name
form.elements[*i*]
form.elements['*имя*']

Свойства

Объект Submit наследует свойства от Input и HTMLElement и определяет или перекрывает следующие:

`value`

Доступная только для чтения строка, задающая текст, который появляется внутри кнопки Submit. Текст задается атрибутом `value` тега `<input>`, создавшего кнопку.

Если атрибут value не указан, по умолчанию value равно «Submit Query» или другой аналогичной строке на языке броузера по умолчанию. В броузерах, которые не умеют переформатировать содержимое документа, это свойство может быть доступно только для чтения.

Методы

Submit наследует методы от Input и HTMLElement.

Обработчик событий

Submit наследует обработчики событий от Input и HTMLElement и определяет или перекрывает следующий:

onclick

Вызывается при щелчке по кнопке Submit.

Синтаксис HTML

Объект Submit создается с помощью стандартного тега HTML <input>:

```
<form>
    ...
  <input
    type="submit"            // Указывает, что это кнопка Submit
    [ value="метка" ]        // Текст, который должен появиться внутри кнопки
                             // Задает значение свойства value
    [ name="имя" ]           // Имя, по которому впоследствии можно ссылаться на кнопку
                             // Задает значение свойства name
    [ onclick="обработчик" ] // Инструкции JavaScript, подлежащие исполнению
                             // при щелчке по кнопке
  >
    ...
</form>
```

Объекты Submit могут также создаваться с помощью тега HTML 4 <button>:

```
<button id="имя"
        type="submit"
        value="значение"
        onclick="обработчик">метка
</button>
```

Описание

Элемент Submit поддерживает те же свойства и методы, что и объект Button, но имеет более специальное назначение. Когда выполняется щелчок по кнопке Submit, выполняется отправка данных формы, содержащей эту кнопку, на сервер, заданный атрибутом action формы, и загружается результирующая HTML-страница, полученная от сервера. В JavaScript 1.1 исключение состоит в том, что форма не отправляется, если либо обработчик события Submit.onclick, либо обработчик события Form.onsubmit вернул false.

Обратите внимание, что в JavaScript 1.1 метод Form.submit() предоставляет альтернативный способ передачи данных формы.

Если для объекта Submit атрибут value не указан, то он обычно помечается текстом «Submit Query». В некоторых формах имеет смысл использовать в качестве метки кнопки текст «Сохранить», «Готово» или «Отправить».

См. также

Button, Form.onsubmit, Form.submit(), HTMLElement, Input; HTMLInputElement в «Справочнике по W3C DOM»

Submit.onclick

JavaScript 1.0; расширен в JavaScript 1.1

вызывается при щелчке по кнопке Submit

Синтаксис

```
<input type="submit" onclick="обработчик" ... >
submit.onclick
```

Описание

Свойство `onclick` объекта Submit ссылается на функцию обработки события, вызываемую, когда пользователь щелкает по кнопке Submit. Подробности см. на справочной странице «HTMLElement.onclick». Однако обратите внимание, что `Submit.onclick` поддерживается начиная с JavaScript 1.0, в отличие от обобщенного обработчика `HTMLElement.onclick`.

Кнопка Submit реализует специальную функцию отправки формы на сервер. Обработчик события `onclick` может добавить к кнопке Submit любую дополнительную функциональность. В JavaScript 1.1 обработчик `onclick` может возвращать `false` для предотвращения передачи данных формы объектом Submit. (Например, обработчик `onclick` может выполнять проверку формы и возвращать `false`, если в форме не заполнены обязательные поля.)

См. также

Form.onsubmit, Form.submit(), HTMLElement.onclick; главу 19; EventListener, EventTarget и MouseEvent в «Справочнике по W3C DOM»

Text

JavaScript 1.0; расширен в JavaScript 1.1

графическое поле ввода текста **наследуется от Input, HTMLElement**

Синтаксис

```
form.имя
form.elements[i]
```

Свойства

Объект Text наследует свойства от Input и HTMLElement и определяет или перекрывает следующее:

`value`

Доступная для чтения и записи строка, задающая текст, который отображается в поле ввода текста. Этот текст может быть введен пользователем или являться значением по умолчанию, заданным в документе или в сценарии. Начальное значение этого свойства задается атрибутом `value` тега `<input>`, который определяет объект Text. Когда пользователь вводит символы в объект Text, свойство `value` обновляется в соответствии с пользовательским вводом. Если устанавить значение свойства `value` явно, то указанная строка отображается в объекте Text. Это свойство также задает строку, отправляемую на сервер при передаче данных формы.

Методы

Объект Text наследует методы Input и HTMLElement.

Обработчик событий

Объект Text наследует обработчики Input и HTMLElement и определяет или перекрывает следующий:

onchange

Вызывается, когда пользователь изменяет значение в элементе Text и перемещает клавиатурный фокус в другое место. Этот обработчик не вызывается для каждого нажатия клавиши в элементе Text, а только когда пользователь завершит редактирование.

Синтаксис HTML

Элемент Text создается с помощью стандартного тега HTML <input>:

```
<form>
   ...
  <input
    type="text"                 // Указывает, что это элемент Text
    [ name="имя" ]              // Имя, по которому можно будет обращаться к этому элементу
                                // Задает значение свойства name
    [ value="по_умолчанию" ]    // Значение по умолчанию, передаваемое при передаче формы
                                // Задает значение свойства defaultValue
    [ size="целое" ]            // Ширина элемента в символах
    [ maxlength="целое" ]       // Максимально разрешенное количество вводимых символов
    [ onchange="обработчик" ]   // Обработчик события onchange()
  >
   ...
</form>
```

Описание

Элемент Text представляет текстовое поле ввода в форме. Атрибут size задает ширину поля ввода на экране, а атрибут maxlength задает максимальное количество символов, которое пользователю разрешено ввести.

Кроме этих атрибутов HTML наибольший интерес в элементе Text представляет свойство value. Это свойство можно читать для получения введенных пользователем данных или установить его, чтобы показать произвольный (неформатированный) текст в поле ввода.

Применение

Если значение, запрашиваемое у пользователя, является секретной информацией, например паролем, который не должен быть показан на экране в открытом виде, то надо применять элемент Password, а не Text. Возможность ввести несколько строк текста предоставляется пользователю при помощи элемента Textarea.

Форма, содержащая только один элемент Text или Password, автоматически передается, если пользователь нажимает в этом элементе Text или Password клавишу Return. Для многих форм это полезно. Однако в некоторых случаях нажатие пользователем клавиши Return и передача данных формы до ввода информации в другие элементы формы, такие как Checkbox и Radio, может ввести его в заблуждение. От этой путаницы можно частично избавиться, помещая передаваемые по умолчанию текстовые элементы в нижней части формы.

См. также

Form, Input, Password, Textarea; HTMLInputElement в «Справочнике по W3C DOM»

Text.onchange

JavaScript 1.0

обработчик, вызываемый при изменении введенного значения

Синтаксис

```
<input type="text" onchange="обработчик" ... >
text.onchange
```

Описание

Свойство onchange элемента Text ссылается на функцию обработки события, которая вызывается, когда пользователь изменяет значение в поле ввода и затем «фиксирует» эти изменения, переместив клавиатурный фокус (то есть переместив курсор мыши или нажав клавишу Tab или Return).

Обратите внимание, что обработчик события onchange *не* вызывается, если значение свойства value объекта Text устанавливается из JavaScript. Также заметьте, что этот обработчик предназначен для обработки завершенного изменения введенных данных и потому не вызывается при каждом нажатии клавиши. Информацию о получении уведомления о каждом событии нажатия клавиши см. на справочной странице «HTMLElement.onkeypress».

Оработчик событий onchange рассмотрен в справочной статье «Input.onchange».

См. также

HTMLElement.onkeypress, Input.onchange; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Textarea

JavaScript 1.0; расширен в JavaScript 1.1

многострочная область ввода **наследуется от Input, HTMLElement**

Синтаксис

```
form.имя
form.elements[i]
```

Свойства

Объект Textarea наследует свойства Input и HTMLElement и определяет или перекрывает следующее:

value

Доступное для чтения и записи свойство. Начальное значение этого свойства совпадает со значением свойства defaultValue: обычный текст (без тегов HTML), присутствующий между тегами <textarea> и </textarea>. Когда пользователь вводит символы в объект Textarea, свойство value обновляется в соответствии с введенными данными. Если свойство value устанавливается явно, то указанная строка отображается в объекте Textarea. Это свойство value содержит строку, отправляемую на сервер при передаче данных формы.

Методы

Textarea наследует методы Input и HTMLElement.

Обработчик событий

Textarea наследует обработчики событий Input и HTMLElement и определяет или перекрывает следующий:

onchange

Вызывается, когда пользователь изменяет значение в элементе Textarea и перемещает клавиатурный фокус в другое место. Этот обработчик события вызывается не для каждого нажатия клавиши в элементе Textarea, а только когда пользователь завершает редактирование.

Синтаксис HTML

Textarea создается с помощью стандартных тегов HTML <textarea> и </textarea>:

```
<form>
    ...
  <textarea
    [ name="имя" ]              // Имя, по которому можно ссылатьсяна этот элемент
    [ rows="целое" ]            // Высота элемента в строках
    [ cols="целое" ]            // Ширина элемента в символах
    [ onchange="обработчик" ]   // Обработчик события onchange()
  >
    обычный_текст               // Первоначальный текст; задает defaultValue
  </textarea>
    ...
</form>
```

Описание

Элемент Textarea представляет текстовое поле ввода в форме. Атрибут name задает имя элемента. Он предоставляет удобный способ обращения к элементу Textarea из кода JavaScript и является обязательным, если форма должна отправляться на сервер. Атрибут cols задает ширину элемента в символах при его выводе на экран, а атрибут rows задает высоту элемента в строках текста. Атрибут wrap указывает, как следует обращаться с длинными строками: если он принимает значение off, то такие строки надо оставлять без изменений; если значение virtual, то они должны отображаться с переводами строк, но передаваться без них; если значение physical, то они должны отображаться и передаваться со вставленными переводами строк.

Кроме этих атрибутов HTML наибольший интерес в элементе Textarea представляет свойство value. Его можно читать для получения введенных пользователем данных или установить его, чтобы показать произвольный (неформатированный) текст в поле ввода. Начальное значение свойства value (и постоянное значение свойства defaultValue) – это текст, присутствующий между тегами <textarea> и </ textarea>.

Обратите внимание, что объект Textarea – это особый вид объекта Input, и он наследуется от Input, хотя объекты Textarea создаются не с помощью тегов HTML <input>.

Применение

Для того чтобы получить только одну строку для ввода текста, используйте элемент Text. Если вводимый текст является секретной информацией, например паролем, то нужен элемент Password.

См. также

Form, HTMLElement, Input, Password, Text; HTMLTextAreaElement в «Справочнике по W3C DOM»

Textarea.onchange

JavaScript 1.0

обработчик, вызываемый при изменении введенных данных

Синтаксис

```
<textarea onchange="обработчик" ... >
  ...
</textarea>
textarea.onchange
```

Описание

Свойство `onchange` элемента Textarea ссылается на функцию обработки события, вызываемую, когда пользователь изменяет значение в текстовой области и затем «фиксирует» эти изменения, переместив клавиатурный фокус в другое место.

Обратите внимание, что обработчик `onchange` *не* вызывается, когда значение свойства `value` объекта Text устанавливается из JavaScript. Также обратите внимание, что этот обработчик предназначен для обработки завершенного изменения во введенном значении и потому не вызывается при каждом нажатии клавиши. Информацию о получении уведомления о каждом событии нажатия клавиши см. на справочной странице «HTMLElement.onkeypress».

Подробности об обработчике `onchange` см. в статье «Input.onchange».

См. также

HTMLElement.onkeypress, Input.onchange; главу 18; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

URL

См. Link, Location или Document.URL

Window

JavaScript 1.0; расширен в JavaScript 1.1 и 1.2

окно или фрейм веб-броузера

Синтаксис

```
self
window
window.frames[i]
```

Свойства

Объект Window определяет следующие свойства. Несовместимые, специфические для броузеров свойства перечислены отдельно после этого списка.

`closed`

Доступное только для чтения свойство, указывающее, было ли окно закрыто. Когда окно броузера закрывается, представляющий его объект Window не исчезает просто так. Объект Window продолжает существовать, но его свойство `closed` устанавливается равным `true`.

`defaultStatus`

Доступная для чтения и записи строка, задающая сообщение, по умолчанию выводимое в строку состояния. См. справочную страницу «Window.defaultStatus».

`document`

Доступная только для чтения ссылка на объект Document, который описывает документ, содержащийся в этом окне или фрейме. Подробности см. в описании объекта Document.

`frames[]`

Массив объектов Window, по одному на каждый фрейм, содержащийся в этом окне. Свойство `frames.length` содержит количество элементов в массиве `frames[]`. Обратите внимание, что фреймы, на которые ссылается массив `frames[]`, могут сами содержать фреймы и могут иметь свой массив `frames[]`.

`history`

Доступная только для чтения ссылка на объект History данного окна или фрейма. Подробности см. в описании объекта History.

`length`

Количество фреймов, содержащихся в этом окне или фрейме. `length` также задает количество элементов в массиве `frames[]`.

`location`

Объект Location для этого окна или фрейма. Этот объект задает URL текущего загруженного документа. Установка этого свойства равным новой строке URL приводит к загрузке и отображению содержимого этого URL в броузере. Более подробную информацию можно найти в описании объекта Location.

`Math`

Ссылка на объект, содержащий различные математические функции и константы. Подробности см. в описании объекта Math.

`name`

Строка, содержащая имя окна. Имя может быть задано, когда окно создается с помощью метода `open()`. В JavaScript 1.0 доступно только для чтения; доступно для чтения и записи в JavaScript 1.1. См. справочную страницу «Window.name».

`navigator`

Доступная только для чтения ссылка на объект Navigator, предоставляющая информацию о версии и конфигурации веб-броузера. Подробности см. в описании объекта Navigator.

`opener` *[JavaScript 1.1]*

Доступная для чтения и записи ссылка на объект Window, в котором содержится сценарий, вызвавший метод `open()` для открытия этого окна броузера верхнего уровня. Это свойство действительно только для объектов Window, представляющих окна верхнего уровня, но не для объектов, представляющих фреймы. Свойство `opener` полезно для того, чтобы созданное окно могло ссылаться на переменные и функции, определенные в создавшем его окне.

`parent`

Доступная только для чтения ссылка на объект Window, содержащий это окно или фрейм. Если это окно является окном верхнего уровня, `parent` ссылается на

само окно. Если это окно является фреймом, свойство `parent` ссылается на окно или фрейм, в котором содержится данное окно.

`screen` *[JavaScript 1.2]*

Объект Screen, разделяемый всеми окнами в броузере. Этот объект Screen содержит свойства, в которых указана информация об экране: число доступных пикселов и доступных цветов. Подробности см. в описании объекта Screen.

`self`

Доступная только для чтения ссылка на само окно. Синоним свойства `window`.

`status`

Доступная для чтения и записи строка, задающая текущее содержимое строки состояния броузера. Подробности см. на справочной странице «Window.status».

`top`

Доступная только для чтения ссылка на окно верхнего уровня, содержащее данное окно. Если данное окно само является окном верхнего уровня, `top` просто содержит ссылку на само окно. Если данное окно представляет собой фрейм, свойство `top` ссылается на окно верхнего уровня, содержащее данный фрейм. Сравните со свойством `parent`.

`window`

Свойство `window` идентично свойству `self`; оно содержит ссылку на это окно.

Свойства в Netscape

`innerHeight, innerWidth` *[Netscape 4]*

Доступные для чтения и записи свойства, задающие высоту и ширину в пикселах экранной области отображения этого окна. Эти размеры не включают высоту панели меню, полос прокрутки и т. д. По соображениям безопасности не допускается устанавливать любое из этих свойств равным значению, меньшему 100 пикселов.

`java` *[Netscape 3]*

Ссылка на объект JavaPackage, являющийся вершиной иерархии имен пакетов для базовых пакетов *java.**, составляющих язык Java. См. справочную страницу «JavaPackage».

`locationbar.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, отображает ли окно адресную панель. См. описание элементов окна на справочной странице «Window.open()».

`menubar.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, отображает ли окно панель меню. См. описание элементов окна на справочной странице «Window.open()».

`netscape` *[Netscape 3]*

Ссылка на объект JavaPackage, являющийся вершиной иерархии имен пакетов для Java-пакетов *netscape.** от Netscape. См. справочную страницу «JavaPackage».

`outerHeight, outerWidth` *[Netscape 4]*

Доступные для чтения и записи целые свойства, задающие общую высоту и ширину окна в пикселах. Эти размеры включают высоту и ширину панели меню, панелей инструментов, полос прокрутки, границ окна и т. д.

`Packages` *[Netscape 3]*

Ссылка на объект JavaPackage, представляющий вершину иерархии имен Java-пакетов. Например, для ссылки на пакет *java.lang* используйте `Packages.java.lang`. См. справочную страницу «JavaPackage».

`pageXOffset, pageYOffset` *[Netscape 4]*

Доступное только для чтения целое, задающее число пикселов, на которые текущий документ был прокручен вправо (`pageXOffset`) и вниз (`pageYOffset`).

`personalbar.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, отображается ли в этом окне «персональная панель» закладок. См. описание элементов окна на справочной странице «Window.open()».

`screenX, screenY` *[Netscape 4]*

Доступные только для чтения целые, задающие координаты X и Y верхнего левого угла окна на экране. Если окно является фреймом, эти свойства задают координаты X и Y окна верхнего уровня, в котором содержится фрейм.

`scrollbars.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, видны ли полосы прокрутки в этом окне или будут ли они видны, если документ будет настолько велик или настолько широк, что они потребуются. Это свойство фактически указывает, включен ли скроллинг для этого окна. См. описание элементов окна на справочной странице «Window.open()».

`statusbar.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, имеется ли в этом окне строка состояния. См. описание элементов окна на справочной странице «Window.open()».

`sun` *[Netscape 3]*

Ссылка на объект JavaPackage, являющийся вершиной иерархии имен Java-пакетов *sun.** от Sun Microsystems. См. справочную страницу «JavaPackage».

`toolbar.visible` *[Netscape 4]*

Доступное только для чтения логическое значение, указывающее, отображается ли в окне панель инструментов. См. описание элементов окна на справочной странице «Window.open()».

Свойства в Internet Explorer

`clientInformation` *[IE 4]*

Специфический для IE синоним свойства `navigator`. Оба свойства ссылаются на объект Navigator. Несмотря на то что `clientInformation` — это более подходящее имя и менее привязано к Netscape, чем `navigator`, оно не поддерживается в Netscape и потому не является переносимым.

`event` *[IE 4]*

Объект Event, содержащий детали о самом последнем произошедшем в окне событии. В событийной модели Netscape 4 и стандартной событийной модели объект Event, описывающий событие, передается в качестве аргумента каждому обработчику событий. Однако в модели IE объект Event не передается, и обработчики должны получать информацию о событии из свойства `event` объекта Window.

Методы

В объекте Window имеются следующие переносимые методы. Несовместимые, специфические для броузера методы перечислены отдельно после этого списка.

`alemrt()`
: Отображает простое сообщение в диалоговом окне.

`blur()`
: Принимает клавиатурный фокус от окна броузера верхнего уровня; на большинстве платформ это приводит к переводу объекта на задний план.

`clearInterval()`
: Отменяет периодическое исполнение кода.

`clearTimeout()`
: Отменяет действие времени ожидания.

`close()`
: Закрывает окно.

`confirm()`
: Задает в диалоговом окне вопрос «да или нет».

`focus()`
: Передает клавиатурный фокус окну броузера верхнего уровня; на большинстве платформ это приводит к переводу окна на передний план.

`moveBy()`
: Перемещает окно на относительную позицию.

`moveTo()`
: Перемещает окно на абсолютную позицию.

`open()`
: Создает и открывает новое окно.

`print()`
: Имитирует щелчок по кнопке Print броузера. Только в IE 5 и Netscape 4.

`prompt()`
: Запрашивает простую строку в диалоговом окне.

`resizeBy()`
: Изменяет размер окна на указанную величину.

`resizeTo()`
: Изменяет размер окна на указанный.

`scroll()`
: Прокручивает документ, отображаемый в окне.

`scrollBy`
: Прокручивает окно на указанную величину.

`scromllTo()`
: Прокручивает окно к указанной позиции.

`setInterval()`
: Исполняет код через указанные промежутки времени.

`setTimeout()`
: Исполняет код по истечении указанного промежутка времени.

Методы в Netscape

`back()`

Ведет себя так же, как если бы пользователь щелкнул по кнопке Back.

`captureEvents()`

Задает типы событий, которые должны непосредственно направляться в указанное окно.

`forward()`

Имитирует щелчок по кнопке Forward броузера.

`handleEvent()`

Вызывает соответствующий обработчик события для данного объекта Event.

`home()`

Отображает домашнюю страницу броузера.

`releaseEvents()`

Задает типы событий, которые больше не подлежат перехвату.

`routeEvent()`

Передает объект Event в соответствующий обработчик следующего заинтересованного в нем объекта.

`stop()`

Имитирует щелчок по кнопке Stop броузера.

Методы в Internet Explorer

`navigate()`

Загружает и отображает указанный URL.

Обработчики событий

`onblur`

Вызывается, когда окно теряет фокус.

`onerror`

Вызывается в случае возникновения ошибки JavaScript.

`onfocus`

Вызывается, когда окно получает фокус.

`onload`

Вызывается, когда документ (или набор фреймов) загружен полностью.

`onmove`

Вызывается при перемещении окна. Только в Netscape 4.

`onresize`

Вызывается при изменении размера окна.

`onunload`

Вызывается, когда броузер выходит из текущего документа или набора фреймов.

Описание

Объект Window представляет окно или фрейм броузера. Он подробно описан в главе 13. В клиентском JavaScript объект Window выступает в качестве «глобально-

го объекта», и все выражения вычисляются в контексте текущего объекта Window. Это значит, что для обращения к текущему окну не требуется специального синтаксиса, и свойства этого объекта можно использовать, как если бы они были глобальными переменными. Например, можно писать `document` вместо *`window`*`.document`. Аналогично, можно вызывать методы текущего объекта окна, как если бы они были функциями, например `alert()` вместо *`window`*`.alert()`.

В объекте Window имеются свойства `window` и `self`, которые ссылаются на само окно. Они позволяют явно указывать ссылку на окно. В дополнение к этим двум свойствам свойства `parent`, `top` и `frames[]` ссылаются на другие объекты Window, имеющие отношение к текущему окну.

Так можно обратиться к фрейму в окне:

```
frames[i] или self.frames[i]  // Фреймы текущего окна
window.frames[i]              // Фреймы указанного окна
```

А к родительскому окну (или фрейму) данного фрейма – так:

```
parent или self.parent  // Родительское окно (фрейм) текущего окна
window.parent           // Родительское окно (фрейм) указанного окна
```

Нижеследующий код позволяет обратиться к окну броузера верхнего уровня из любого фрейма, содержащегося в нем:

```
top или self.top  // Окно верхнего уровня для текущего фрейма
window.top        // Окно верхнего уровня для указанного фрейма
```

Новые окна верхнего уровня создаются с помощью метода `Window.open()`. Вызывая этот метод, сохраните возвращаемое при вызове `open()` значение в переменной, чтобы иметь ссылку на новое окно. В JavaScript 1.1 свойство `opener` нового окна является ссылкой на открывшее его окно.

Как правило, методы объекта Window выполняют какие-либо манипуляции с окном или фреймом броузера. Стоит отметить методы `alert()`, `confirm()` и `prompt()`; они служат для взаимодействия с пользователем через простые диалоговые окна.

Всесторонний обзор объекта Window приведен в главе 13, а подробности обо всех свойствах, методах и обработчиках событий можно найти на соответствующих справочных страницах.

См. также

Document; главу 13; AbstractView в «Справочнике по W3C DOM»

Window.alert() JavaScript 1.0

отображает сообщение в диалоговом окне

Синтаксис

`window``.alert(`*`сообщение`*`)`

Аргументы

сообщение

Текстовая (не HTML) строка, которая должна быть показана в диалоговом окне, всплывающем над окном *`window`*.

Описание

Метод `alert()` показывает пользователю указанное сообщение в диалоговом окне. Диалоговое окно содержит кнопку OK, по которой пользователь может щелкнуть, чтобы закрыть диалоговое окно.

На платформах Windows метод `alert()` отображает модальное диалоговое окно, и исполнение кода JavaScript приостанавливается до тех пор, пока пользователь это окно не закроет. Однако в Netscape 4 на платформах Unix диалоговое окно `alert()` не является модальным, и исполнение продолжается без перерыва.

Применение

Пожалуй, чаще всего метод `alert()` применяется для вывода сообщений об ошибке, когда пользователь вводит в какой-либо элемент некорректные данные. Предупреждающее окно может информировать пользователя об ошибке и объяснять, что должно быть исправлено, чтобы избежать ошибок в будущем.

Внешний вид диалогового окна `alert()` зависит от платформы, но, как правило, оно содержит графическое изображение, обозначающее какую-либо ошибку или предупреждение. Хотя `alert()` может выводить любое желаемое сообщение, предупреждающее графическое изображение диалогового окна означает, что этот метод не подходит для простых информационных сообщений вроде «Добро пожаловать на мою домашнюю страницу» или «Вы – 177-й посетитель за эту неделю!».

Обратите внимание, что *сообщение*, выводимое в диалоговом окне, – это строка текста, а не форматированный HTML. В сообщении может присутствовать символ перевода строки «\n», разбивающий его на несколько строк. Можно выполнять некоторое простейшее форматирование с помощью пробелов и имитировать горизонтальные линии с помощью символов подчеркивания, но результат сильно зависит от шрифта в диалоговом окне и потому в разных системах выглядит по-разному.

См. также

Window.confirm(), Window.prompt()

Window.back()

Netscape 4

возврат к предыдущему документу

Синтаксис

window.back()

Описание

Вызов `back()` приводит к тому, что броузер показывает документ, который ранее был показан в окне *window*, точно так же, как если бы пользователь щелкнул по кнопке Back этого окна.

Обратите внимание, что в документах с фреймами поведение методов `Window.back()` и `History.back()` может отличаться.

Window.blur()

JavaScript 1.1

убирает клавиатурный фокус из окна верхнего уровня

Синтаксис

window.blur()

Описание

Метод `blur()` убирает клавиатурный фокус от окна броузера верхнего уровня, задаваемого объектом Window. Если объект Window представляет собой фрейм, то клавиатурный фокус передается окну верхнего уровня, содержащего данный фрейм. На большинстве платформ при выводе клавиатурного фокуса из окна верхнего уровня оно переводится на задний план (то есть вниз стека окон).

См. также

Window.focus()

Window.captureEvents() Netscape 4

задает типы событий, которые должны перехватываться

Синтаксис

window.captureEvents(*маска_событий*)
document.captureEvents(*маска_событий*)
layer.captureEvents(*маска_событий*)

Аргументы

маска_событий

Целое значение, задающее тип событий, которые должны перехватываться окном, документом или слоем. Это значение должно быть равным одной из статических констант типов событий, определенных в классе Event, или быть группой констант типов событий, объединяемых с помощью операторов поразрядного ИЛИ (|) или сложения.

Описание

`captureEvents()` — это метод классов Window, Document и Layer. Его назначение для всех трех классов одинаково: в событийной модели Netscape 4 он определяет, что все события данного типа или типов, происходящие в указанном окне, документе или слое, должны передаваться в окно, документ или слой, а не в объект, в котором они фактически происходят.

Тип событий, которые должны быть перехвачены, задается аргументом *маска_события*, битовой маской, составленной из статических констант, определенных в классе Event. Полный список этих констант можно найти на справочной странице «Event.TYPE».

См. также

Event, Window.handleEvent(), Window.releaseEvents(), Window.routeEvent(); главу 19; EventTarget.addEventListener() в «Справочнике по W3C DOM»

Window.clearInterval() JavaScript 1.2

останавливает периодическое исполнение кода

Синтаксис

window.clearInterval(*intervalId*)

Аргументы

intervalId

Значение, возвращаемое при соответствующем вызове `setInterval()`.

Описание

`clearInterval()` останавливает периодическое исполнение кода, которое было начато с помощью вызова `setInterval()`. *intervalId* должно быть значением, возвращенным при вызове метода `setInterval()`.

См. также

Window.setInterval()

Window.clearTimeout() JavaScript 1.0

отменяет отложенное исполнение

Синтаксис

window.`clearTimeout(`*timeoutId*`)`

Аргументы

timeoutId

Значение, возвращаемое методом `setTimeout()`, которое идентифицирует отменяемое действие.

Описание

`clearTimeout()` отменяет исполнение кода, которое было отложено с помощью метода `setTimeout()`. Аргумент *timeoutId* — это значение, возвращаемое при вызове `setTimeout()` и идентифицирующее один или несколько блоков отменяемого кода.

См. также

Window.setTimeout()

Window.close() JavaScript 1.0

закрывает окно броузера

Синтаксис

window.`close()`

Описание

Метод `close()` закрывает окно броузера верхнего уровня, задаваемое окном *window*. Окно может закрыть само себя с помощью вызова `self.close()` или просто `close()`.

В JavaScript 1.1 с помощью JavaScript могут быть закрыты только те окна, которые были открыты с помощью JavaScript. Благодаря этому злонамеренные сценарии не могут закрывать броузер пользователя.

Не существует разумного способа закрыть фрейм в окне. Следовательно, метод `close()` должен вызываться только для объектов Window, представляющих окна броузера верхнего уровня, а не для объектов, представляющих фреймы.

См. также

Window.open(), свойства `closed` и `opener` объекта Window

Window.confirm() JavaScript 1.0

задает вопрос «да-или-нет»

Синтаксис

`window``.confirm(`*`вопрос`*`)`

Аргументы

вопрос

Обычная текстовая (не HTML) строка, выводимая в диалоговом окне. Обычно она представляет собой вопрос, который вы хотите задать пользователю.

Возвращаемое значение

`true`, если пользователь щелкнул по кнопке OK; `false`, если пользователь щелкнул по кнопке Cancel.

Описание

Метод `confirm()` выводит указанный *вопрос* в диалоговом окне, всплывающем над окном *`window`*. Внешний вид диалогового окна зависит от платформы, но, как правило, оно содержит графическое изображение, обозначающее, что пользователю задается вопрос. Диалоговое окно содержит кнопки OK и Cancel, позволяющие на этот вопрос ответить. Если была нажата кнопка OK, `confirm()` возвращает `true`. Если кнопка Cancel, `confirm()` возвращает `false`.

Метод `confirm()` выводит *модальное* диалоговое окно, то есть окно, блокирующее любой ввод пользователя в главном окне броузера до тех пор, пока пользователь не закроет диалоговое окно, щелкнув по кнопке OK или Cancel. Данный метод возвращает значение в зависимости от отклика пользователя, поэтому исполнение кода JavaScript приостанавливается на вызове `confirm()` и последующие инструкции не исполняются, пока пользователь не нажмет одну из кнопок в диалоговом окне.

Применение

Обратите внимание, что *вопрос*, выводимый в диалоговом окне, представляет собой строку обычного текста, а не форматированный HTML. В нем допускается наличие символа перевода строки «\n», разбивающего сообщение на несколько строк. Можно выполнять некоторое простейшее форматирование с помощью пробелов и имитировать горизонтальные линии с помощью символов подчеркивания, но результат сильно зависит от шрифта в диалоговом окне и потому в разных системах выглядит поразному.

Кроме того, нет способа изменить метки, появляющиеся на кнопках диалогового окна (чтобы, например, в них было написано «Да» и «Нет»). Следовательно, необходимо позаботиться о том, чтобы на вопрос можно было ответить либо «OK», либо «Cancel».

См. также

Window.alert(), Window.prompt()

Window.defaultStatus

JavaScript 1.0

текст строки состояния по умолчанию

Синтаксис

window.defaultStatus

Описание

defaultStatus — это доступное для чтения и записи строковое свойство, задающее текст по умолчанию, который будет появляться в строке состояния окна. Обычно веб-броузеры отображают в строке состояния хода загрузки файла и выводят в нее адреса гиперссылок, над которыми проходит курсор мыши. Пока в строке состояния не выводятся эти временные сообщения, она по умолчанию пуста. Однако в свойстве defaultStatus можно задать сообщение, выводимое в строку состояния по умолчанию, а прочитав значение свойства defaultStatus, можно узнать это сообщение по умолчанию. Задаваемый текст может быть временно переписан другими сообщениями, например теми, которые выводятся, когда пользователь перемещает курсор мыши над гиперссылкой, но сообщение defaultStatus всегда выводится заново при удалении временного сообщения.

Если установить defaultStatus для объекта Window, являющегося фреймом, то заданное сообщение будет видно всегда, когда курсор мыши находится внутри фрейма (независимо от наличия фокуса у этого фрейма). Если defaultStatus задано для окна верхнего уровня, не содержащего фреймов, то сообщение видно всегда, когда видимо окно. Если задать defaultStatus для окна верхнего уровня, содержащего фреймы, то сообщение будет отображаться, только когда курсор мыши находится над границами, разделяющими фреймы. Следовательно, чтобы гарантировать видимость сообщения в документе, содержащем фреймы, необходимо установить defaultStatus для всех фреймов в документе.

Применение

defaultStatus применяется для вывода полупостоянных сообщений в строке состояния. Временное сообщение выводится при помощи свойства status.

См. также

Window.status

Window.focus()

JavaScript 1.1

передает клавиатурный фокус окну верхнего уровня

Синтаксис

window.focus()

Описание

Метод focus() передает клавиатурный фокус окну броузера верхнего уровня, заданному объектом Window. Если объект Window представляет собой фрейм, то клавиатурный фокус передается фрейму и окну верхнего уровня, его содержащему.

На большинстве платформ окно верхнего уровня при получении фокуса перемещается в стеке окон наверх.

См. также

Window.blur()

Window.forward() Netscape 4

перемещает вперед к следующему документу

Синтаксис

window.forward()

Описание

Вызов forward() заставляет броузер показать в окне *window* следующий документ, точно так же, как если бы пользователь щелкнул по кнопке Forward.

Обратите внимание, что в документах, содержащих фреймы, методы Window.forward() и History.forward() ведут себя по-разному.

Window.handleEvent() Netscape 4

передает событие соответствующему обработчику

Синтаксис

window.handleEvent(*событие*)
document.handleEvent(*событие*)
layer.handleEvent(*событие*)
htmlElement.handleEvent(*событие*)

Аргументы

событие

Объект события, которое должно быть обработано.

Возвращаемое значение

Любое значение, возвращаемое обработчиком события, который вызывается для обработки *события*.

Описание

handleEvent() – это метод классов Window, Document и Layer и всех элементов HTML, поддерживающих обработчики событий. При вызове для любого объекта *o* метод handleEvent() определяет тип своего аргумента *событие* и передает этот объект Event соответствующему обработчику в объекте *o*.

См. также

Window.routeEvent(); главу 19

Window.home() Netscape 4

отображает домашнюю страницу

Синтаксис

window.home()

Описание

Вызов home() приводит к отображению броузером домашней страницы, как если бы пользователь щелкнул по кнопке Home броузера.

Window.moveBy() JavaScript 1.2

перемещает окно на относительную позицию

Синтаксис

window.moveBy(*dx*, *dy*)

Аргументы

dx – количество пикселов, на которые окно должно переместиться вправо.

dy – количество пикселов, на которые окно должно переместиться вниз.

Описание

moveBy() перемещает *окно* на относительную позицию, заданную аргументами *dx* и *dy*. По соображениям безопасности броузеры могут ограничивать действие сценариев, запрещая им перемещать окно за пределы экрана.

Window.moveTo() JavaScript 1.2

перемещает окно на абсолютную позицию

Синтаксис

window.moveTo(*x*, *y*)

Аргументы

x – координата X новой позиции окна.

y – координата Y новой позиции окна.

Описание

moveTo() перемещает окно *window* таким образом, чтобы его верхний левый угол находился в позиции, заданной аргументами *x* и *y*. По соображениям безопасности, броузеры могут ограничивать действие этого метода, запрещая ему перемещать окно за пределы экрана.

Window.name JavaScript 1.0; в JavaScript 1.1 доступно для чтения и записи

имя окна

Синтаксис

window.name

Описание

Свойство *name* – это строка, задающая имя окна *window*. Это свойство доступно только для чтения в JavaScript 1.0 и для чтения/записи в JavaScript 1.1. Имя окна верхнего уровня первоначально задается в аргументе *name* метода Window.open(). Имя фрейма первоначально задается атрибутом name тега HTML <frame>.

Имя окна верхнего уровня или фрейма может выступать в качестве значения атрибута target тегов <a> или <form>. В этом случае документ, на который указывает ссылка, или результат передачи данных формы должен быть отображен в окне с данным именем.

Исходное окно, открытое броузером, и любые окна, открытые через пункт меню New Web Browser, изначально не имеют имени (то есть name == ""), следовательно, к этим окнам нельзя обратиться с помощью атрибута target из другого отдельного окна верх-

него уровня. В JavaScript 1.1 можно установить атрибут name для исправления этой ситуации.

См. также

Form.target, Link.target

Window.navigate() Internet Explorer 3

загружает новый URL

Синтаксис

window.navigate(*url*)

Аргументы

url

Строка, задающая URL, который должен быть загружен и показан.

Описание

Метод Window.navigate() в Internet Explorer загружает указанный *url* в окно *window* («перемещается» к *url*).

Метод navigate() не поддерживается в Netscape. То же действие может быть выполнено в Netscape и IE присваиванием требуемого *url* свойству location требуемого окна.

См. также

Location, свойство location объекта Window

Window.onblur JavaScript 1.1

обработчик, вызываемый при потере окном клавиатурного фокуса

Синтаксис

```
<body onblur="обработчик" ... >
<frameset onblur="обработчик" ... >
window.onblur
```

Описание

Свойство onblur объекта Window задает функцию обработки события, вызываемую при потере окном клавиатурного фокуса.

Начальное значение этого свойства – функция, содержащая инструкции JavaScript, указанные через точку с запятой в атрибуте onblur тегов <body> или <frameset>.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onblur. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего элемент.

Применение

Обработчик события onblur() может применяться для остановки анимации, когда окно не имеет фокуса ввода. Теоретически, если окно не имеет фокуса, пользователь, скорее всего, не может видеть анимацию или не обращает на нее внимания.

См. также

Window.blur(), Window.focus(), Window.onfocus; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Window.onerror

JavaScript 1.1; имеет ошибку в Netscape 6/6.1

обработчик, вызываемый при возникновении ошибки JavaScript

Синтаксис

Обработчик события `onerror` можно зарегистрировать следующим образом:

```
window.onerror=функция-обработчик
```

Броузер вызывает обработчик так:

```
window.onerror(сообщение, url, строка)
```

Аргументы

сообщение

Строка, задающая сообщение о произошедшей ошибке.

url

Строка, задающая URL документа, в котором произошла ошибка.

строка

Число, задающее номер строки, в которой произошла ошибка.

Возвращаемое значение

`true`, если обработчик выполнил обработку ошибки и JavaScript не должен предпринимать никаких дальнейших действий; `false`, если JavaScript должен вывести стандартное диалоговое окно для этой ошибки.

Описание

Свойство `onerror` объекта Window задает функцию обработки события, вызываемую, когда в коде, исполняемом в этом окне, возникает ошибка JavaScript. По умолчанию, когда возникает ошибка, JavaScript выводит диалоговое окно. Программист может сам реализовать обработку ошибок, написав собственный обработчик `onerror`.

Можно определить обработчик ошибки `onerror` для окна, установив свойство `onerror` объекта Window равным соответствующей функции. Обратите внимание, что в отличие от других обработчиков ошибок JavaScript, обработчик `onerror` не может быть определен в теге HTML.

Вызываемому обработчику `onerror` передаются три аргумента: строка, задающая сообщение об ошибке; строка, задающая URL документа, в котором произошла ошибка, и число, задающее номер строки, где произошла ошибка. Функция обработки ошибки может делать с этими аргументами все, что угодно; может, например, вывести собственное сообщение об ошибке или зарегистрировать ее любым другим способом. Завершившись, обработчик ошибок должен вернуть `true`, если обработка ошибки выполнена, и от JavaScript не требуются какие-либо дальнейшие действия; `false`, если обработчик только заметил или как-то зарегистрировал ошибку, и все еще требуется, чтобы JavaScript показал сообщение об ошибке в стандартном диалоговом окне.

Обратите внимание, что если этот обработчик события возвращает `true`, сообщая броузеру о том, чтобы он не предпринимал дальнейших действий, то большинство обра-

ботчиков объекта Form и элементов формы возвращают false, чтобы не дать броузеру выполнить какие-либо действия, например подтвердить форму. Эта непоследовательность может ввести в заблуждение.

Можно полностью отключить обработку ошибок в окне, присвоив свойству onerror окна функцию, возвращающую true и не выполняющую ничего. Можно восстановить стандартную обработку ошибок (диалоговое окно), присвоив onerror функцию, возвращающую false и ничего не делающую.

Ошибки

Этот обработчик события корректно вызывается при возникновении ошибок в Netscape 6 и Netscape 6.1, но значения, передаваемые в аргументах *сообщение*, *url* и *строка* не корректны, поэтому несмотря на то, что он позволяет обнаружить факт ошибки, с его помощью нельзя получить какую-либо полезную информацию об этой ошибке.

Window.onfocus JavaScript 1.1

обработчик, вызываемый при получении окном фокуса

Синтаксис

```
<body onfocus="обработчик" ... >
<frameset onfocus="обработчик" ... >
window.onfocus
```

Описание

Свойство onfocus объекта Window задает функцию обработки события, вызываемую, когда окно получает клавиатурный фокус.

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, указанные через точку с запятой в атрибуте onfocus тега <body> или <frameset>.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onfocus. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего элемент.

Применение

Обработчик события onfocus можно применять для запуска анимации, а обработчик onblur — для ее остановки, чтобы анимация работала, только когда пользователь обращает внимание на окно.

См. также

Window.blur(), Window.focus(), Window.onblur; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Window.onload

JavaScript 1.0

обработчик, вызываемый по завершении загрузки документа

Синтаксис

```
<body onload="обработчик" ... >
<frameset onload="обработчик" ... >
window.onload
```

Описание

Свойство onload объекта Window задает функцию обработки события, которая вызывается, когда документ или набор фреймов целиком загружен в свое окно или фрейм.

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, заданные через точку с запятой в атрибуте onload тегов <body> или <frameset>.

Если вызван обработчик события onload, можно быть уверенными, что документ загружен целиком и, следовательно, что все сценарии в документе выполнены, все функции в сценариях определены, а все формы и другие элементы документа уже проанализированы и доступны через объект Document.

Применение

Если какой-либо из обработчиков событий в вашем документе зависит от того, был ли документ загружен полностью, необходимо проверять этот факт перед выполнением этих обработчиков. Если сетевое соединение было остановлено после появления кнопки в документе, но до того, как загрузились части документа, на которые полагается эта кнопка, то после щелчка по кнопке броузер поведет себя незапланированным образом или сгенерирует сообщение об ошибке. Один из хороших способов, позволяющих убедиться, что документ загружен, состоит в том, чтобы использовать обработчик onload для установки значения переменной (например, loaded) и проверять значение этой переменной перед выполнением чего-либо, что зависит от полной загрузки документа.

См. также

Window.onunload; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Window.onmove

Netscape 4; не поддерживается в Netscape 4 на платформах Unix

обработчик, вызываемый при перемещении окна

Синтаксис

```
<body onmove="обработчик" ... >
<frameset onmove="обработчик" ... >
window.onmove
```

Описание

Свойство onmove объекта Window задает функцию обработки событий, вызываемую, когда пользователь перемещает окно верхнего уровня на новую позицию на экране.

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, указанные через точку с запятой в атрибуте onmove тега <body> или

<frameset>. Если функция обработки события определяется в атрибуте HTML, то она исполняется в области действия данного объекта Window, а не родительского окна.

В качестве аргумента функции обработки события onmove передается объект Event. Свойства этого объекта содержат информацию о новых координатах окна.

Window.onresize

JavaScript 1.2

обработчик, вызываемый при изменении размера окна

Синтаксис

```
<body onresize="обработчик" ... >
<frameset onresize="обработчик" ... >
window.onresize
```

Описание

Свойство onresize объекта Window задает функцию обработки ошибок, вызываемую, когда пользователь изменяет размер окна или фрейма.

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, заданные через точку с запятой в атрибуте onresize тега <body> или <frameset>. Если функция обработки события определяется в атрибуте HTML, то она исполняется в области действия данного объекта Window, а не родительского окна.

В событийной модели Netscape 4 объект Event передается в качестве аргумента функции-обработчику onresize. В событийной модели IE аргумент не передается, но соответствующий объект Event доступен в качестве свойства event объекта Window, содержащего элемент.

В Netscape новый размер окна доступен из свойств width и height объекта Event.

Window.onunload

JavaScript 1.0

обработчик, вызываемый, когда броузер покидает страницу

Синтаксис

```
<body onunload="обработчик" ... >
<frameset onunload="обработчик" ... >
window.onunload
```

Описание

Свойство onunload объекта Window задает функцию обработки событий, вызываемую, когда броузер «выгружает» документ или набор фреймов, готовясь к загрузке нового документа.

Начальное значение этого свойства представляет собой функцию, содержащую инструкции JavaScript, заданные через точку с запятой в атрибуте onunload тегов <body> или <frameset>. Обработчик события onunload предоставляет возможность полностью обновить состояние броузера перед загрузкой нового документа.

Если броузер покидает сайт, использующий фреймы, обработчик onunload набора фреймов вызывается до обработчика onunload каждого из фреймов. Это порядок, обратный порядку вызова обработчиков onload.

Обработчик onunload() вызывается, когда пользователь дает броузеру команду покинуть текущую страницу и переместиться на другую. Следовательно, в обычных обстоятельствах нежелательно задерживать загрузку запрошенной страницы, генерируя всплывающие диалоговые окна (например, с помощью Window.confirm() или Window.prompt()) из обработчика события onunload.

См. также

Window.onload; главу 19; Event, EventListener и EventTarget в «Справочнике по W3C DOM»

Window.open()

JavaScript 1.0; расширен в JavaScript 1.1

открывает новое окно броузера или находит окно с указанным именем

Синтаксис

window.open(*url*, *имя*, *элементы*, *замена*)

Аргументы

url

Необязательная строка, задающая URL, который должен отображаться в новом окне. Если этот аргумент опущен или задана пустая строка, то в новом окне никакой документ не показывается.

имя

Необязательная строка алфавитно-цифровых символов и символов подчеркивания, задающая имя для нового окна. Это имя может выступать в качестве значения атрибута target тегов HTML <a> и <form>. Если этот аргумент указывает на уже существующее окно, метод open() не создает нового окна, а просто возвращает ссылку на окно с этим именем. В этом случае аргумент *элементы* игнорируется.

элементы

Строка, указывающая, какие элементы стандартного окна броузера должны присутствовать в новом окне. Формат этой строки приведен в разделе «Элементы окна». Этот аргумент может отсутствовать, и тогда новое окно будет иметь все стандартные графические элементы.

замена

Необязательный логический аргумент, указывающий, должен ли URL, загруженный в новое окно, создавать новую запись в истории окна или заменять текущую запись в истории просмотра. Если этот аргумент равен true, новая запись истории не создается. Этот аргумент был добавлен в JavaScript 1.1. Обратите внимание, что не имеет особого смысла задавать этот аргумент для только что созданных окон; он предназначен для изменения содержимого существующего окна.

Возвращаемое значение

Ссылка на объект Window, который может быть либо только что созданным, либо новым окном, в зависимости от аргумента *имя*.

Описание

Метод open() ищет существующее или открывает новое окно броузера. Если аргумент *имя* задает имя существующего окна, то возвращается ссылка на это окно. Возвращаемое окно отображает URL, указанный в аргументе *url*, а аргумент *элементы* игнори-

руется. В JavaScript это единственный способ, позволяющий получить ссылку на окно, для которого известно только имя.

Если аргумент *имя* не указан или если окно с этим именем еще не существует, метод open() создает новое окно броузера. Созданное окно отображает URL, заданный аргументом *url*, имеет имя, заданное аргументом *имя*, и размеры и управляющие элементы, заданные аргументом *элементы* (формат этого аргумента описан в следующем разделе). Если *url* – это пустая строка, open() открывает пустое окно.

Аргумент *имя* задает имя нового окна. Это имя может содержать только алфавитно-цифровые символы и символы подчеркивания. Оно может выступать в качестве значения атрибута target тега <a> или <form> в HTML для вывода документов в это окно.

В JavaScript 1.1 в случае применения Window.open() для загрузки нового документа в окно с указанным именем можно передать аргумент *замена*, указывающий, должна ли для нового документа создаваться новая запись в истории просмотра окна или документ должен заменить текущую запись в истории окна. Если аргумент *замена* равен true, то новый документ заменяет старый. Если этот аргумент равен false или не указан, то новый документ создает новую запись в истории просмотра окна. Этот аргумент предоставляет функциональность, во многом схожую с методом Location.replace().

Не путайте Window.open() с методом Document.open() – эти два метода выполняют совершенно разные функции. Для ясности следует отказаться от open() в пользу Window.open(). В обработчиках событий, определенных как атрибуты HTML, open() обычно интерпретируется как Document.open(), поэтому в таких случаях необходимо вызывать Window.open().

Элементы окна

Аргумент *элементы* – это список элементов окна, указанных через запятую, которые должны появиться в окне. Если этот необязательный аргумент пуст или не указан, то в окне будут присутствовать все элементы. С другой стороны, если в аргументе *элементы* указан хотя бы один элемент, любые элементы, не указанные в списке, будут в окне отсутствовать. Строка не может содержать пробелы или другие пробельные символы. Каждый элемент в списке имеет следующий формат:

элемент[=*значение*]

Для большинства элементов *значение* может быть равно yes или no. Для этих элементов знак равенства и *значение* могут опускаться – если элемент указан, подразумевается yes, если не указан – no. Для элементов width и height *значение* является обязательным и должно задавать размер окна в пикселах.

Доступные элементы и их значения:

channelmode

Указывает, должно ли окно быть показано в режиме канала. Только в IE 4.

dependent

Если значение равно «no», указывает, что новое окно не должно быть зависимым дочерним окном текущего окна. Только в Netscape 4.

directories

Кнопки каталогов, такие как «What's New» и «What's Cool». Только в Netscape.

fullscreen

Указывает, должно ли окно быть показано в полноэкранном режиме. Только в IE 4.

`height`

Задает высоту области отображения окна в пикселах.

`innerHeight`

Задает высоту области документа окна в пикселах. Только в Netscape 4.

`innerWidth`

Задает ширину области документа окна в пикселах. Только в Netscape 4.

`left`

Координата X окна в пикселах. Только в IE 4. В Netscape используйте `screenX`.

`location`

Поле ввода для набора URL непосредственно в броузере.

`menubar`

Панель меню.

`outerHeight`

Задает общую высоту окна в пикселах. Только в Netscape 4.

`outerWidth`

Задает общую ширину окна в пикселах. Только в Netscape 4.

`resizable`

Если этот элемент не указан или равен `no`, окно не имеет вокруг своих границ меток для изменения размера. (В зависимости от платформы пользователь все равно может иметь другую возможность изменения размера окна.) Обратите внимание на распространенную ошибку в написании этого элемента – «resizeable» – с лишней буквой «е».

`screenX`

Координата X окна в пикселах. Только в Netscape 4. В IE 4 используйте `left`.

`screenY`

Координата Y окна в пикселах. Только в Netscape 4. В IE 4 используйте `top`.

`scrollbars`

Включает горизонтальные и вертикальные полосы прокрутки, когда они необходимы.

`status`

Строка состояния.

`toolbar`

Панель инструментов броузера с кнопками Back, Forward и т. д.

`top`

Координата Y окна в пикселах. Только в IE 4. В Netscape используйте `screenY`.

`width`

Задает ширину области отображения документа в пикселах.

См. также

Location.replace(), Window.close(), свойства `closed` и `opener` объекта Window

Window.print()

Netscape 4, Internet Explorer 5

печатает документ

Синтаксис

window.print()

Описание

print() вызывает печать текущего документа точно так же, как при щелчке пользователя по кнопке Print броузера.

Window.prompt()

JavaScript 1.0

позволяет получить строку, введенную в диалоговом окне

Синтаксис

window.prompt(*сообщение*, *по умолчанию*)

Аргументы

сообщение

Текстовая (не HTML) строка, которая должна быть показана в диалоговом окне. Она должна просить пользователя ввести необходимую информацию.

по умолчанию

Строка, отображаемая в диалоговом окне по умолчанию. Для того чтобы метод prompt() показал пустое окно ввода, надо передать пустую строку ("").

Возвращаемое значение

Строка, вводимая пользователем; пустая строка, если пользователь не ввел строку; или null, если пользователь щелкнул по кнопке Cancel.

Описание

Метод prompt() выводит указанное *сообщение* в диалоговом окне, которое также содержит поле ввода текста и кнопки OK, Clear и Cancel. Зависимое от платформы графическое изображение в диалоговом окне указывает пользователю, что от него требуется ввод данных.

Если пользователь щелкает по кнопке Cancel, метод prompt() возвращает null. Если пользователь щелкает по кнопке Clear, prompt() удаляет любой текущий текст в поле ввода. Если пользователь щелкает по кнопке OK, prompt() возвращает значение, отображаемое в данный момент в поле ввода.

Метод prompt() отображает *модальное* диалоговое окно, блокирующее любой пользовательский ввод в главном окне броузера до тех пор, пока пользователь не избавится от диалогового окна, щелкнув по кнопке OK или Cancel. Данный метод возвращает значение, зависящее от выбора пользователя в диалоговом окне, поэтому исполнение JavaScript приостанавливается на время вызова prompt(), и последующие инструкции не исполняются, пока пользователь не нажмет кнопку в диалоговом окне.

См. также

Window.alert(), Window.confirm()

Window.releaseEvents() Netscape 4

прекращает перехват событий

Синтаксис

window.releaseEvents(*маска_событий*)
document.releaseEvents(*маска_событий*)
layer.releaseEvents(*маска_событий*)

Аргументы

маска_событий

Целое, задающее тип событий, перехват которых в окне, документе или слое должен быть прекращен. Это значение должно быть одной из статических констант типов событий, определенных в классе Event, или быть группой констант типов событий, объединенных с помощью оператора поразрядного ИЛИ (|) или сложения.

Описание

Метод releaseEvents() объектов Window, Document и Layer выполняет действие, противоположное выполняемому методом captureEvents() этих классов. В событийной модели Netscape 4 releaseEvents() указывает, что окно, документ или слой больше не должны перехватывать события типов, заданных *маскойсобытий*. Список констант, которые могут использоваться в аргументе *маскасобытий*, можно найти на справочной странице «Event.TYPE».

См. также

Event, Window.captureEvents(), Window.handleEvent(), Window.routeEvent(); главу 19; EventTarget.removeEventListener() в «Справочнике по W3C DOM»

Window.resizeBy() JavaScript 1.2

изменяет размер окна на относительное значение

Синтаксис

window.resizeBy(*dw*, *dh*)

Аргументы

dw — количество пикселов, на которые должна быть изменена ширина окна.

dh — количество пикселов, на которые должна быть изменена высота окна.

Описание

resizeBy() изменяет размер окна *window* на относительное значение, заданное аргументами *dh* и *dw*. По соображениям безопасности броузер может ограничивать этот метод, не позволяя устанавливать высоту или ширину окна меньшими, чем 100 пикселов.

Window.resizeTo() JavaScript 1.2

изменяет размер окна

Синтаксис

window.resizeTo(*ширина*, *высота*)

Аргументы

ширина

Желаемое значение ширины окна.

высота

Желаемое значение высоты окна.

Описание

Метод `resizeTo()` изменяет размер окна *`window`* так, чтобы он имел указанную *ширину* и *высоту*. По соображениям безопасности броузер может ограничивать этот метод, не позволяя устанавливать высоту или ширину окна меньшими, чем 100 пикселов.

Window.routeEvent()

Netscape 4

передает перехваченное событие следующему обработчику

Синтаксис

`window``.routeEvent(`*`событие`*`)`
`document``.routeEvent(`*`событие`*`)`
`layer``.routeEvent(`*`событие`*`)`

Аргументы

событие

Перехваченный объект Event, подлежащий передаче следующему обработчику событий.

Возвращаемое значение

Значение, возвращенное обработчиком, в который было отправлено событие *`event`*.

Описание

`routeEvent()` – это метод классов Window, Document и Layer, и он ведет себя одинаково для всех трех классов. Когда перехваченный объект *`event`* передается в обработчик события окна, документа или слоя, этот обработчик может передать событие следующему заинтересованному обработчику событий (если такой существует). Если окно, документ или слой содержит какие-либо другие окна (фреймы), документы или слои, также воспользовавшиеся `captureEvents()` для регистрации заинтересованности в событиях этого типа, событие перенаправляется к соответствующему обработчику такого объекта окна, документа или слоя.

С другой стороны, если ни один из дочерних объектов Window, Document или Layer не выразил интерес к этому событию, `routeEvent()` передает объект *`event`* в соответствующий обработчик события объекта, от которого поступило событие. Комбинация методов `captureEvents()` и `routeEvent()` образует основу событийной модели «просачивания» в Netscape 4.

См. также

Window.captureEvents(), Window.handleEvent(), Window.releaseEvents(); главу 19

Window.scroll() JavaScript 1.1; признан устаревшим в JavaScript 1.2

прокручивает документ в окне

Синтаксис

window.scroll(*x*, *y*)

Аргументы

x – координата X, на которую выполняется прокрутка.

y – координата Y, на которую выполняется прокрутка.

Описание

Метод scroll() перемещает документ в окне так, чтобы указанные *x*- и *y*-координаты документа оказались в левом верхнем углу окна.

Координата X увеличивается в направлении вправо, а координата Y увеличивается в направлении вниз страницы. Иначе говоря, вызов scroll(0,0) всегда помещает верхний левый угол документа в верхний левый угол окна.

В JavaScript 1.2 методу scroll() следует предпочесть scrollTo() и scrollBy().

Window.scrollBy() JavaScript 1.2

прокручивает документ на относительную величину

Синтаксис

window.scrollBy(*dx*, *dy*)

Аргументы

dx – количество пикселов, на которые документ должен быть прокручен вправо.

dy – количество пикселов, на которые документ должен быть прокручен вниз.

Описание

scrollBy() прокручивает документ, отображаемый в окне, на относительную величину, заданную аргументами *dx* и *dy*.

Window.scrollTo() JavaScript 1.2

прокручивает документ

Синтаксис

window.scrollTo(*x*, *y*)

Аргументы

x Координата X, в которой должен оказаться левый край области отображения документа в окне.

y Координата Y, в которой должен оказаться верхний край области отображения документа в окне.

Описание

scrollTo() прокручивает документ, отображаемый в окне *window*, так, чтобы точка в документе, заданная координатами *x* и *y*, если это возможно, отображалась в левом верхнем углу.

Метод `scrollTo()` является предпочтительным по сравнению с методом `Window.scroll()` JavaScript 1.1, который делает то же самое, но имеет неадекватно описывающее его функцию имя.

Window.setInterval() JavaScript 1.2; IE 4 поддерживает только одну из двух форм

периодически выполняет указанный код

Синтаксис

window.setInterval(*код*, *интервал*)
window.setInterval(*функция*, *интервал*, *аргументы...*)

Аргументы

код

Строка кода JavaScript, подлежащая периодическому выполнению. Если строка содержит несколько инструкций, то они должны отделяться друг от друга точками с запятой.

функция

Функция JavaScript, подлежащая периодическому выполнению. Эта форма метода не доступна в IE 4.

интервал

Целое значение, задающее интервал в миллисекундах между вызовами *кода* или *функции*.

аргументы...

Любое количество произвольных значений, передаваемых в качестве аргументов при каждом вызове *функции*.

Возвращаемое значение

Значение, которое может быть передано в метод `Window.clearInterval()` для отмены периодического исполнения *кода* или *функции*.

Описание

`setInterval()` периодически исполняет инструкции JavaScript, заданные в строке *код* через *интервал* миллисекунд.

В Netscape 4, но не в IE 4, в качестве первого аргумента вместо строки может быть передана функция. В этой форме метода `setInterval()` указанная *функция* периодически повторяется через *интервал* миллисекунд. Любые дополнительные значения *аргументов*, переданные методу `setInterval()`, передаются в качестве аргументов при каждом вызове *функции()*.

В обеих формах метод `setInterval()` возвращает значение, которое затем может быть передано методу `Window.clearInterval()` для прекращения периодического исполнения *кода* или *функции*.

`setInterval()` сходен с `setTimeout()`. Последний применяется тогда, когда надо отложить исполнение кода так, чтобы он не запускался периодически.

См. также

Window.clearInterval(), Window.setTimeout()

Window.setTimeout() JavaScript 1.0

отложить исполнение кода

Синтаксис

window.setTimeout(*код*, *задержка*)

Аргументы

код

Строка, содержащая код JavaScript, который должен быть исполнен по истечении указанного промежутка *задержка*.

задержка

Пауза в миллисекундах перед выполнением инструкций JavaScript в строке аргумента *код*.

Возвращаемое значение

Загадочное значение («идентификатор времени ожидания»), которое может быть передано в метод clearTimeout() для отмены исполнения *кода*.

Описание

Метод setTimeout() откладывает исполнение инструкций JavaScript в строке *код* на количество миллисекунд, указанное в аргументе *задержка*. По истечении указанного количества миллисекунд инструкции в *коде* исполняются обычным образом. Обратите внимание, что они должны исполняться только один раз. Для периодического исполнения кода сам *код* должен содержать вызов метода setTimeout(), регистрирующий повторное исполнение кода. В JavaScript 1.2 можно регистрировать код, исполняемый через установленные интервалы, посредством метода Window.setInterval().

Инструкции в строке *код* исполняются в контексте окна *window*, то есть *window* для этих инструкций является текущим окном. Если в строке *код* присутствует несколько инструкций, их необходимо разделить точками с запятой.

См. также

Window.clearTimeout(), Window.setInterval()

Window.status JavaScript 1.0

задает временное сообщение в строке состояния

Синтаксис

window.status

Описание

status – это строковое свойство, доступное для чтения и записи, задающее временное сообщение, которое должно появиться в строке состояния окна. Как правило, сообщение появляется только на ограниченное время – пока оно не будет переписано другим сообщением или пока пользователь не переместит курсор мыши в другую область окна. Когда сообщение, заданное в свойстве status, стирается, строка состояния возвращается к своему пустому по умолчанию состоянию или к сообщению, заданному в свойстве defaultStatus.

Несмотря на то что только окна верхнего уровня имеют строки состояния, можно также устанавливать свойство status во фреймах. Тогда указанное сообщение выводится в строке состояния окна верхнего уровня. Временные сообщения, установленные фреймами, отображаются независимо от того, какой фрейм имеет фокус в данный момент или в каком фрейме находится курсор мыши. Это поведение отличается от поведения свойства defaultStatus.

Применение

Свойство status применяется для вывода временных сообщений в строку состояния. Для вывода полупостоянных сообщений предназначено свойство defaultStatus.

Как правило, установка свойства status имеет смысл только из обработчиков событий и фрагментов кода, исполнение которых отложено с помощью метода Window.setTimeout(). Если значение status устанавливается непосредственно из сценария, то пользователь не увидит сообщение. Оно не отображается сразу, а когда будет показано, то наверняка будет моментально переписано сообщением броузера, например «Document: done».

Для того чтобы установить свойство status в обработчике событий onmouseover гиперссылки, необходимо вернуть true из этого обработчика событий, потому что когда курсор мыши проходит над ссылкой, действием по умолчанию является отображение URL этой ссылки, что приводит к стиранию любого сообщения в строке состояния, установленного обработчиком события. Вернув true из обработчика события, вы отменяете это действие по умолчанию и сохраняете ваше собственное сообщение status в строке состояния (пока курсор мыши не уйдет из ссылки).

См. также

Window.defaultStatus

Window.stop()

Netscape 4

останавливает загрузку документа

Синтаксис

window.stop()

Описание

Вызов stop() останавливает загрузку броузером текущего документа точно так, как при щелчке пользователя по кнопке Stop броузера.

Часть V. Справочник по W3C DOM

Это полный справочник по всем объектам, свойствам, функциям, методам и обработчикам событий JavaScript-реализации W3C DOM. На первых нескольких страницах поясняется, как работать с этим справочным материалом.

Справочник по W3C DOM

В этом справочнике описаны интерфейсы, методы и свойства, определенные в стандартах W3C DOM Level 1 и Level 2. Программисты со средним или большим опытом, пишущие для новейшего поколения веб-броузеров, совместимых со стандартами, найдут этот справочник полезным дополнением к справочникам по базовому и клиентскому JavaScript в частях III и IV. Введение и пример справочной страницы поясняют, как сделать работу с этим справочником максимально эффективной. Между этим и двумя другими справочниками есть существенные различия, и данное введение надо прочитать внимательно.

Данный справочник, как и два других, организован в алфавитном порядке. Справочные страницы, посвященные методам и свойствам интерфейсов DOM, отсортированы по их полным именам, включающим имена определяющих интерфейсов. Так, описание метода `appendChild()` интерфейса Node надо искать по заголовку «Node.appendChild», а не по «appendChild».

В целях экономии места в этом справочнике нет собственных справочных страниц для свойств (однако такие страницы есть для всех интерфейсов и методов). Вместо этого каждое свойство полностью документируется на справочной странице интерфейса, в котором оно определено. Например, о свойстве `tagName` интерфейса Element можно прочитать на справочной странице «Element».

Может оказаться, что вы не помните имени интерфейса, определяющего метод или свойство, или не знаете точно, в каком из трех справочников искать класс или интерфейс. Часть VI этой книги представляет собой специальный указатель, призванный помочь в таких ситуациях. Найдите имя класса, интерфейса, метода или свойства и из указателя вы узнаете, в каком из трех справочников и под каким классом надо искать нужную информацию. Так, если вы ищете «Document», то заглянув в указатель части VI, вы увидите, что статьи с этим именем имеются и в справочнике по клиентскому JavaScript, и в справочнике по DOM. А разыскивая имя «firstChild», узнаете, что `firstChild` – это свойство интерфейса Node, о котором можно прочитать в справочнике по DOM.

Найти нужную информацию на справочной странице не составит особого труда. Однако поскольку стандарт DOM предназначен для работы с языками, отличными от JavaScript, при его создании принимались в расчет типизированные языки (такие как Java и C++). Несмотря на то что JavaScript –

это нетипизированный язык, определенная в стандарте DOM информация о типе свойств и методов полезна и включена в данный справочник. Это означает, что в разделе «Синтаксис» для свойств и методов используется язык, более похожий на Java, чем на JavaScript. Далее следует пример справочной страницы, озаглавленный «Пример справочной статьи», который демонстрирует структуру каждой справочной страницы и поясняет, как интерпретировать информацию, представленную в каждом разделе. Даже если вы уже хорошо знакомы с третьим изданием этой книги, не пожалейте времени на прочтение этого примера, прежде чем начать работу со «Справочником по W3C DOM».

Пример справочной статьи

доступность

как читать справочные страницы по DOM **наследуется от**

Заголовок и краткое описание

Каждая справочная статья начинается с состоящего из четырех частей блока заголовка, подобного приведенному выше. Статьи отсортированы по заголовку. Краткое описание, приведенное под заголовком, дает представление о предмете, рассмотренном в данной статье; оно поможет быстро решить, представляет ли для вас интерес остальная часть справочной страницы.

Доступность

Информация о доступности показана в правом верхнем углу блока заголовка и сообщает, какой уровень и какой модуль стандарта DOM определяет данный интерфейс или метод. Свойства не имеют своих собственных справочных страниц, поэтому информация о доступности для них отсутствует. Если доступность свойства отлична от доступности определяющего его интерфейса, этот факт отмечается в описании свойства.

Наследуется от

Интерфейсы DOM могут наследовать свойства и методы от других интерфейсов. Если интерфейс DOM является наследником другого интерфейса, иерархия наследования показывается в правом нижнем углу блока заголовка. Например, информация «наследуется от» для элемента HTMLElement выглядит следующим образом:

Node → Element → HTMLElement

Это означает, что HTMLElement наследуется от интерфейса Element, который, в свою очередь, является наследником интерфейса Node. Благодаря этой информации мы знаем, какие справочные страницы посвящены родственным интерфейсам.

Подынтерфейсы

В этом разделе содержится информация, противоположная информации «наследуется от»: в ней перечислены любые интерфейсы, наследующиеся от данного. Например, раздел «Подынтерфейсы» справочной страницы интерфейса Element указывает, что HTMLElement является подынтерфейсом Element и наследует методы и свойства интерфейса Element.

Также реализует

Из модульной структуры стандарта DOM следует, что некоторые интерфейсы разбиты на несколько отдельных, чтобы конкретные реализации JavaScript могли содер-

жать только те интерфейсы, которые являются частью поддерживаемых ими модулей. Часто объект, реализующий один интерфейс (например, Document), также реализует несколько других простых интерфейсов (например, DocumentCSS, DocumentEvent и DocumentViews), чтобы обеспечить функциональность, относящуюся к другим модулям. Если интерфейс имеет дополнительные интерфейсы, предназначенные для реализации совместно с ним, они перечисляются в этом разделе.

Константы

Некоторые интерфейсы DOM определяют набор констант, выступающих в качестве значений для свойства или в качестве аргументов метода этого интерфейса. Например, интерфейс Node определяет важные константы, служащие в качестве набора допустимых значений свойства `nodeType` всех узлов документа. Если интерфейс определяет константы, они перечисляются и документируются в этом разделе. Список включает тип, имя, значение (именно в этом порядке) каждой константы. Синтаксис, используемый в этих списках, описан в разделе «Типы DOM». Обратите внимание, что константы являются статическими свойствами самого интерфейса, а не экземпляров этого интерфейса.

Свойства

Если справочная страница описывает интерфейс, то в этом разделе перечисляются свойства, определяемые этим интерфейсом. Каждая запись в этом списке содержит имя и тип свойства и может также включать другие ключевые слова, предоставляющие дополнительную информацию об этом свойстве. Обратите внимание, что в этом Java-подобном синтаксисе имя свойства указывается в конце, а все, что предшествует имени, сообщает тип свойства и прочую информацию о нем. Например, интерфейсы HTMLTableElement и HTMLTableCellElement определяют, в том числе, следующие свойства:

`HTMLTableCaptionElement caption`

Свойство `caption`. Оно ссылается на объект типа HTMLTableCaptionElement.

`readonly HTMLCollection rows`

Свойство `rows`. Ссылается на объект HTMLCollection и доступно только для чтения: значение этого свойства можно прочитать, но его нельзя установить.

`deprecated String align`

Свойство `align`. Это строка, но свойство признано устаревшим, и его использование не поощряется.

`readonly long cellIndex`

Свойство `cellIndex`. Это значение типа «длинное целое» (см. справочную страницу «Типы DOM») и доступно только для чтения.

Методы

Если справочная страница описывает интерфейс, то в этом разделе перечисляются имена методов интерфейса и кратко описывается каждый метод. Полное описание каждого метода приводится на отдельной справочной странице.

Синтаксис

Если справочная страница описывает метод, то в этом разделе представлена сигнатура или синтаксис метода. В разделе употребляется Java-подобный синтаксис и указывается следующая информация (по порядку):

- Тип возвращаемого методом значения или `void`, если метод не возвращает ничего.
- Имя метода.
- Типы и имена (в указанном порядке) всех аргументов метода. Они представлены в виде списка типов и имен аргументов, указанных через запятую в скобках. Если метод не принимает ни одного аргумента, вы просто увидите скобки `()`.
- Типы исключений, если они есть, которые могут генерироваться данным методом.

Например, раздел «Синтаксис» метода `Node.insertBefore()` выглядит так:

```
Node insertBefore(Node новыйДочернийУзел,
                  Node ДочернийУзел)
    throws DOMException;
```

Из этого синтаксиса можно извлечь следующую информацию: имя метода – «insertBefore»; возвращаемое значение – объект Node; первый аргумент – объект Node, задающий «новыйДочернийУзел» (предположительно, вставляемый узел); второй аргумент – также объект Node, задающий «ДочернийУзел» (предположительно, узел, перед которым вставляется другой узел); кроме того, при некоторых обстоятельствах метод может генерировать исключение типа DOMException.

Подразделы, следующие за «Синтаксисом», содержат дополнительную информацию об аргументах, возвращаемом значении и исключениях метода. В разделе «Типы DOM» можно найти дополнительную информацию о Java-подобном синтаксисе, используемом здесь для задания типов аргументов метода.

Аргументы

Если метод имеет аргументы, за разделом «Синтаксис» следует подраздел «Аргументы», в котором перечислен и описан каждый из аргументов. Обратите внимание, что имена аргументов выделены *курсивом*, то есть вместо них подставляются некоторые значения или выражения JavaScript. Продолжая предыдущий пример, подраздел «Аргументы» статьи `Node.insertBefore()` выглядит следующим образом:

новыйДочернийУзел

Узел, вставляемый в дерево. Если это DocumentFragment, вместо него вставляются его дочерние элементы.

ДочернийУзел

Дочерний узел данного узла, перед которым должен быть вставлен *новыйДочернийУзел*. Если этот аргумент равен `null`, то *новыйДочернийУзел* вставляется в качестве последнего дочернего узла данного.

Возвращаемое значение

Раздел «Синтаксис» задает тип данных возвращаемого методом значения, а подраздел «Возвращаемое значение» содержит дополнительную информацию. Если метод не возвращает никакого значения (то есть если в разделе «Синтаксис» он указан как возвращающий `void`), этот раздел опускается.

Исключения

В этом подразделе описаны виды исключений, которые могут генерироваться этим методом, и обстоятельства, при которых они генерируются.

Типы DOM

В этом справочнике для указания типов констант, свойств, возвращаемых методами значений и аргументов методов, употребляется Java-подобный синтаксис. В данном

разделе представлена дополнительная информация об этом синтаксисе. Обратите внимание, что сами справочные страницы не имеют разделов «Типы DOM»!

Общий синтаксис таков:

```
модификаторы тип имя
```

Имя константы, свойства, метода или аргумента метода всегда приводится в конце и ему предшествует тип данных и другая информация. Вот модификаторы, используемые в этом справочном разделе (обратите внимание, что на самом деле они не являются допустимыми модификаторами Java):

`readonly`

Указывает, что значение свойства может быть прочитано, но не может быть установлено.

`deprecated`

Указывает, что свойство устарело и его следует избегать.

`unsigned`

Указывает, что числовая константа, свойство, возвращаемое значение или аргумент метода является беззнаковым, то есть могут быть равны нулю или быть положительными, но не могут быть отрицательными.

Типы констант, свойств, возвращаемых методами значений и аргументов методов DOM не всегда соответствуют типам, поддерживаемым в JavaScript. Например, некоторые свойства имеют тип `short`, обозначающий 16-разрядное целое. Несмотря на то что в JavaScript имеется только один числовой тип, в этом справочнике тип DOM используется только потому, что он предоставляет больше информации о допустимом диапазоне значений. Вот типы DOM, которые встречаются в этом справочнике:

`String`

Базовый объект String из JavaScript.

`Date`

Базовый объект Date из JavaScript (используется редко).

`boolean`

Логическое значение: `true` или `false`.

`short`

Короткое (16-разрядное) целое. К этому типу может применяться модификатор `unsigned`.

`long`

Длинное (64-разрядное) целое. К этому типу может применяться модификатор `unsigned`.

`float`

Число с плавающей точкой. К этому типу не может применяться модификатор `unsigned`.

`void`

Этот тип используется только для возвращаемых методами значений; он указывает, что метод не возвращает никакого значения.

Любой другой тип

Любые другие типы, встречающиеся в данном разделе, являются именами других интерфейсов DOM (например, Document, DOMImplementation, Element, HTMLTableElement и Node).

Описание

Большинство справочных страниц содержат раздел «Описание», представляющий собой основное описание документируемого интерфейса или метода. Он является сердцем справочной страницы. При первом знакомстве с интерфейсом или методом можно перейти непосредственно к этому разделу, а затем вернуться и просмотреть предыдущие разделы, такие как «Синтаксис», «Свойства» и «Методы». Если же интерфейс или метод для вас не нов, скорее всего, читать этот раздел не надо, и в поисках определенной информации, например типа аргумента или имени свойства, придется заглянуть в раздел «Свойства» или «Аргументы».

В некоторых статьях данный раздел представляет собой лишь короткий абзац, в других занимает одну или несколько страниц. Для некоторых простых методов разделы «Аргументы», «Возвращаемое значение» и «Исключения» сами по себе достаточно хорошо описывают метод; в этом случае раздел «Описание» опускается.

Пример

Справочные страницы для некоторых часто используемых интерфейсов и методов включают пример, приводимый в данном разделе. Пример иллюстрирует типичное применение интерфейса или метода. Однако большинство страниц не содержат примеров, поскольку их можно найти в первой половине этой книги.

См. также

Большинство справочных страниц завершаются перекрестными ссылками на связанные справочные страницы (в основном «Справочника по W3C DOM»), которые могут представлять интерес. Но некоторые относятся к индивидуальным описаниям свойств, содержащимся внутри справочных страниц интерфейсов, а другие – к связанным справочным страницам в «Справочнике по клиентскому JavaScript» или к первым двум частям книги.

В конце раздела «См. также» справочных страниц, описывающих интерфейсы (но не методы), могут содержаться дополнительные сведения (перекрестные ссылки на способы применения данного интерфейса). Подраздел «Тип» перечисляет свойства, значения которых представляют собой объекты, реализующие интерфейс. В подразделе «Передается в» перечислены методы, принимающие аргументы, которые реализуют этот интерфейс. Подраздел «Возвращается из» содержит список методов, возвращающих объект, который реализует этот интерфейс. Эти перекрестные ссылки показывают, как получить объект этого интерфейса и что можно делать с полученным объектом.

AbstractView

DOM Level 2 Views

окно, отображающее документ

Также реализует

`ViewCSS`

Если реализация DOM поддерживает модуль CSS, то любой объект, реализующий интерфейс AbstractView, также реализует интерфейс ViewCSS. Для удобства метод, определяемый в интерфейсе ViewCSS, также приведен в разделе «Методы».

Свойства

`readonly Document document`

Объект Document, отображаемый данным объектом View. Этот объект Document также реализует интерфейс DocumentView.

Методы

`getComputedStyle()` *[DOM Level 2 CSS]*

Этот метод ViewCSS возвращает доступный только для чтения объект CSSStyleDeclaration, представляющий вычисляемую информацию о стилях для указанного элемента документа.

Описание

В DOM *представление* (*view*) – это объект, каким-либо образом отображающий документ. В качестве примера представления можно привести объект Window клиентского JavaScript. Интерфейс AbstractView – это первый шаг на пути к стандартизации некоторых свойств и методов объекта Window. Он просто указывает, что все объекты View имеют свойство с именем `document`, ссылающееся на документ, который в них отображается. Кроме того, если реализация поддерживает таблицы стилей CSS, то все объекты View также реализуют интерфейс ViewCSS и определяют метод `getComputedStyle()` для задания реального фактического вида элемента в представлении.

Свойство `document` дает любому представлению ссылку на отображаемый им документ. Верно также и обратное: каждый документ имеет ссылку на представление, которое его отображает. Если реализация DOM поддерживает модуль View, то объект, реализующий интерфейс Document, также реализует и интерфейс DocumentView. Этот интерфейс DocumentView определяет свойство `defaultView`, ссылающееся на окно, в котором отображается документ.

Имя этого интерфейса содержит слово «Abstract» (абстрактное), что подчеркивает тот факт, что это лишь начало стандартизации оконного интерфейса. Чтобы извлечь из этого пользу, будущие уровни стандарта DOM должны будут ввести новый интерфейс, расширяющий AbstractView и добавляющий другие свойства или методы.

См. также

Document.defaultView

Тип: Document.defaultView

AbstractView.getComputedStyle() DOM Level 2 CSS

извлекает CSS-стили, применяемые для отображения элемента

Синтаксис

```
CSSStyleDeclaration getComputedStyle(Element elt,
                                     String pseudoElt);
```

Аргументы

elt

Элемент документа, информация о стилях которого извлекается.

pseudoElt

Псевдоэлемент CSS или `null`, если он отсутствует.

Возвращаемое значение

Доступный только для чтения объект CSSStyleDeclaration (обычно также реализующий интерфейс CSS2Properties), содержащий информацию о стилях, которая управляет отображением указанного элемента в данном представлении. Любые размерные значения, запрашиваемые от этого объекта, всегда выражены в абсолютных величинах или в пикселах, а не в процентах или в относительных величинах.

Описание

Элемент документа может получать информацию о стилях из встроенного атрибута `style` или из любого количества таблиц стилей в «каскаде». Прежде чем элемент сможет быть фактически отображен в представлении, его стиль необходимо «вычислить» путем извлечения информации о стилях из соответствующих частей каскада.

Данный метод предоставляет доступ к этим вычисляемым стилям. По контрасту с ним свойство `style` элемента дает доступ только к встроенным стилям элемента и ничего не говорит об атрибутах таблиц стилей, применяемых к элементу. Обратите внимание, что этот метод также предоставляет способ определения реальных координат (в пикселах), где элемент отображается в данном представлении.

Метод `getComputedStyle()` фактически определяется интерфейсом ViewCSS. В любой реализации DOM, поддерживающей модули View и CSS, любой объект, реализующий AbstractView, также всегда реализует ViewCSS. Поэтому для простоты этот метод также указан в интерфейсе AbstractView.

В Internet Explorer аналогичная функциональность доступна через нестандартное свойство `currentStyle` любого объекта HTMLElement.

См. также

CSS2Properties, CSSStyleDeclaration, HTMLElement.style

Attr

DOM Level 1 Core

атрибут элемента документа **Node → Attr**

Свойства

`readonly String name`

Имя атрибута.

`readonly Element ownerElement` *[DOM Level 2]*

Объект Element, содержащий данный атрибут, или `null`, если объект Attr в данный момент не связан ни с каким объектом Element.

`readonly boolean specified`

`true`, если атрибут был явно указан в исходном тексте документа или установлен сценарием. `false`, если атрибут не был задан явно, но для него в DTD документа задано значение по умолчанию.

`String value`

Значение атрибута. При чтении этого свойства значение атрибута возвращается в виде строки. Когда этому свойству присваивается строка, автоматически создается узел Text, содержащий текст этой строки, и этот узел Text становится единственным дочерним узлом объекта Attr.

Описание

Объект Attr представляет атрибут узла Element. Объекты Attr связаны с узлами Element, но не являются непосредственно частью дерева документа (их свойство `parentNode` равно `null`). Объект Attr можно получить из свойства `attributes` интерфейса Node или путем вызова метода `getAttributeNode()` интерфейса Element.

Объекты Attr являются узлами, и значение Attr представлено дочерними узлами узла Attr. В HTML-документах это просто один узел Text. Однако в XML-документах узлы Attr могут иметь дочерние узлы Text и EntityReference. Свойство `value` предоставляет сокращенный способ чтения и записи значения атрибута в виде строки.

В большинстве случаев простейшим способом работы с атрибутами элемента являются методы `getAttribute()` и `setAttribute()` интерфейса Element. Эти методы используют строки в качестве имен и значений атрибута и целиком обходятся без узлов Attr.

См. также

Element

Передается в: Element.removeAttributeNode(), Element.setAttributeNode(), Element.setAttributeNodeNS()

Возвращается из: Document.createAttribute(), Document.createAttributeNS(), Element. getAttributeNode(), Element.getAttributeNodeNS(), Element.removeAttributeNode(), Element.setAttributeNode(), Element.setAttributeNodeNS()

CDATASection

DOM Level 1 XML

раздел CDATA XML-документа — **Node → CharacterData → Text → CDATASection**

Описание

Этот редко применяемый интерфейс представляет раздел CDATA в XML-документе. Программисты, работающие с HTML-документами, никогда не встретят узлов этого типа, и этот интерфейс им не потребуется.

CDATASection – это подынтерфейс интерфейса Text, и он не определяет никаких собственных свойств или методов. Текстовое содержимое раздела CDATA доступно через свойство `nodeValue`, унаследованное от Node, или через свойство `data`, унаследованное от CharacterData. Несмотря на то что узлы CDATASection часто рассматриваются так же, как узлы Text, обратите внимание, что `Node.normalize()` не выполняет слияния соседних разделов CDATA.

См. также

CharacterData, Text

Возвращается из: Document.createCDATASection()

CharacterData

DOM Level 1 Core

общая функциональность узлов Text и Comment — **Node → CharacterData**

Подынтерфейсы

Comment, Text

Свойства

`String data`

Текст, содержащийся в этом узле.

`readonly unsigned long length`

Количество символов, содержащихся в этом узле.

Методы

`appendData()`

Дописывает указанную строку к тексту данного узла.

`deleteData()`

Удаляет из данного узла текст заданной длины, начиная с символа с указанным смещением.

`insertData()`

Вставляет строку в текст узла, начиная с символа с заданным смещением.

`replaceData()`

Заменяет указанное количество символов указанной строкой, начиная с указанного текстового смещения.

`substringData()`

Возвращает копию текста заданной длины, начиная с указанного символа.

Описание

CharacterData – это надынтерфейс для узлов Text и Comment. Документы никогда не содержат узлов CharacterData; они содержат только узлы Text и Comment. Однако поскольку функции, выполняемые узлами этих типов, аналогичны, CharacterData определен в этой реализации, чтобы Text и Comment могли ее наследовать.

См. также

Comment, Text

CharacterData.appendData() DOM Level 1 Core

добавляет строку к узлу Text или Comment

Синтаксис

```
void appendData(String arg)
    throws DOMException;
```

Аргументы

arg

Строка, добавляемая к узлу Text или Comment.

Исключения

Этот метод генерирует DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если он вызывается для узла, доступного только для чтения.

Описание

Этот метод дописывает строку *arg* в конец свойства `data` для данного узла.

CharacterData.deleteData() DOM Level 1 Core

удаляет символы из узла Text или Comment

Синтаксис

```
void deleteData(unsigned long смещение,
                unsigned long количество)
    throws DOMException;
```

Аргументы

смещение

Позиция первого удаляемого символа.

количество

Количество удаляемых символов.

Исключения

Этот метод может генерировать исключение DOMException с одним из следующих значений `code`:

`INDEX_SIZE_ERR`

Если аргумент *смещение* или *количество* отрицателен или если *смещение* больше длины узла Text или Comment.

`NO_MODIFICATION_ALLOWED_ERR`

Узел доступен только для чтения и не может модифицироваться.

Описание

Этот метод удаляет заданное *количество* символов из узла Text или Comment, начиная с символа в позиции *смещение*. Если *смещение* плюс *количество* больше, чем число символов в узле Text или Comment, удаляются все символы, начиная с символа в позиции *смещение* до конца строки.

CharacterData.insertData() DOM Level 1 Core

вставляет строку в узел Text или Comment

Синтаксис

```
void insertData(unsigned long смещение,
                String arg)
    throws DOMException;
```

Аргументы

смещение

Позиция символа внутри узла Text или Comment, куда должна вставляться строка.

arg

Вставляемая строка.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных далее значений `code` в следующих ситуациях:

INDEX_SIZE_ERR

смещение отрицательно или больше длины узла Text или Comment.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может быть изменен.

Описание

Этот метод вставляет указанную строку *arg* в текст узла Text или Comment в указанной позиции *смещение*.

CharacterData.replaceData() DOM Level 1 Core

заменяет символы узла Text или Comment на строку

Синтаксис

```
void replaceData(unsigned long смещение,
                 unsigned long количество,
                 String arg)
    throws DOMException;
```

Аргументы

смещение

Символьная позиция в узле Text или Comment, с которой должна начаться замена.

количество

Количество заменяемых символов.

arg

Строка, заменяющая символы, заданные аргументами *смещение* и *количество*.

Исключения

Этот метод может генерировать исключение DOMException с перечисленными далее значениями code при следующих обстоятельствах:

INDEX_SIZE_ERR

смещение отрицательно или больше, чем длина узла Text или Comment, или *количество* отрицательно.

NO_MODIFICATION_ALLOWED_ERR

Узел доступен только для чтения и не может быть изменен.

Описание

Этот метод заменяет указанное *количество* символов, начиная с позиции *смещение*, в содержимом строки *arg*. Если сумма *смещения* и *количества* больше длины узла Text или Comment, заменяются все символы, начиная с символа в позиции *смещение*.

CharacterData.substringData() DOM Level 1 Core

извлекает подстроку из узла Text или Comment

Синтаксис

```
String substringData(unsigned long смещение,
                     unsigned long количество)
    throws DOMException;
```

Аргументы

смещение

Позиция первого возвращаемого символа.

количество

Число символов в возвращаемой подстроке.

Возвращаемое значение

Строка, состоящая из указанного *количества* символов узла Text или Comment, начиная с позиции *смещение*.

Исключения

Метод может генерировать исключение DOMException с такими значениями `code`:

`INDEX_SIZE_ERR`

смещение отрицательно или больше, чем длина узла Text или Comment, или *количество* отрицательно.

`DOMSTRING_SIZE_ERR`

Указанный диапазон слишком велик, чтобы помещаться в строку в реализации JavaScript в данном броузере.

Описание

Этот метод извлекает подстроку из заданного *количества* символов, начиная с позиции *смещение*, из текста узла Text или Comment. Этот метод полезен, только когда размер содержащегося в узле текста больше максимального числа символов, которые могут быть помещены в строку в реализации JavaScript в данном броузере. В этом случае программа JavaScript не может непосредственно использовать свойство `data` узла Text или Comment и вместо этого должна работать с подстроками меньшего размера из текста узла. На практике такая ситуация маловероятна.

Comment

DOM Level 1 Core

HTML- или XML-комментарий **Node → CharacterData → Comment**

Описание

Узел Comment представляет комментарий в HTML- или XML-документе. Содержимое комментария (то есть текст между `<!--` и `-->`) доступно через свойство `data`, унаследованное от интерфейса CharacterData, или через свойство `nodeValue`, унаследованное от интерфейса Node. Этим содержимым можно манипулировать с помощью различных методов, унаследованных от CharacterData.

См. также

CharacterData

Возвращается из: Document.createComment()

Counter

DOM Level 2 CSS

спецификация CSS для counter() или counters()

Свойства

`readonly String identifier`

Имя счетчика.

`readonly String listStyle`

Стиль списка для счетчика.

`readonly String separator`

Строка-разделитель для вложенных счетчиков.

Описание

Этот интерфейс представляет значение `counter()` или `counters()` в соответствии с CSS. Дополнительную информацию см. в справочнике по CSS.

См. также

Возвращается из: CSSPrimitiveValue.getCounterValue()

CSS2Properties

DOM Level 2 CSS2

вспомогательные свойства для всех атрибутов CSS2

Свойства

В этом интерфейсе определяется большое количество свойств: по одному свойству для каждого атрибута CSS, определяемого спецификацией CSS2. Эти имена свойств в основном соответствуют именам атрибутов CSS, с небольшими изменениями, необходимыми для того, чтобы избежать синтаксических ошибок в JavaScript. Атрибуты, имена которых состоят из нескольких слов и содержат дефисы, такие как «font-family», в JavaScript записываются без дефисов, а каждое слово, кроме первого, пишется с большой буквы: `fontFamily`. Кроме того, атрибут «float» конфликтует с зарезервированным словом `float`, поэтому он транслируется в свойство `cssFloat`.

Полный набор свойств перечислен в следующей таблице. Так как свойства непосредственно соответствуют атрибутам CSS, для всех этих свойств отсутствуют самостоятельные описания. Смысл и допустимые значения атрибутов CSS описаны в литературе по CSS, например в книге издательства O'Reilly «Cascading Style Sheets: The Definitive Guide» Эрика Майера (Eric A. Meyer). Все эти свойства представляют собой строки. Присваивание значений этим свойствам может привести к тем же исключениям и по тем же причинам, что и вызов `CSSStyleDeclaration.setProperty()`.

azimuth	background	backgroundAttachment	backgroundColor
backgroundImage	backgroundPosition	backgroundRepeat	border
borderBottom	borderBottomColor	borderBottomStyle	borderBottomWidth
borderCollapse	borderColor	borderLeft	borderLeftColor
borderLeftStyle	borderLeftWidth	borderRight	borderRightColor
borderRightStyle	borderRightWidth	borderSpacing	borderStyle
borderTop	borderTopColor	borderTopStyle	borderTopWidth
borderWidth	bottom	captionSide	clear
clip	color	content	counterIncrement
counterReset	cssFloat	cue	cueAfter
cueBefore	cursor	direction	display
elevation	emptyCells	font	fontFamily
fontSize	fontSizeAdjust	fontStretch	fontStyle
fontVariant	fontWeight	height	left

letterSpacing	lineHeight	listStyle	listStyleImage
listStylePosition	listStyleType	margin	marginBottom
marginLeft	marginRight	marginTop	markerOffset
marks	maxHeight	maxWidth	minHeight
minWidth	orphans	outline	outlineColor
outlineStyle	outlineWidth	overflow	padding
paddingBottom	paddingLeft	paddingRight	paddingTop
page	pageBreakAfter	pageBreakBefore	pageBreakInside
pause	pauseAfter	pauseBefore	pitch
pitchRange	playDuring	position	quotes
richness	right	size	speak
speakHeader	speakNumeral	speakPunctuation	speechRate
stress	tableLayout	textAlign	textDecoration
textIndent	textShadow	textTransform	top
unicodeBidi	verticalAlign	visibility	voiceFamily
volume	whiteSpace	widows	width
wordSpacing	zIndex		

Описание

Этот интерфейс определяет по одному свойству для каждого атрибута CSS, определенного в спецификации CSS2. Если реализация DOM поддерживает этот интерфейс (являющийся частью модуля «CSS2»), то все объекты CSSStyleDeclaration также реализуют CSS2Properties. Чтение любого из свойств, определенных этим интерфейсом, эквивалентно вызову `getPropertyValue()` для соответствующего атрибута CSS, а установка значения любого из этих свойств эквивалентна вызову `setProperty()` для соответствующего атрибута. Свойства, определенные в CSS2Properties, соответствуют атрибутам-сокращениям CSS, и интерфейс CSS2Properties правильно работает с этими свойствами.

См. также

CSSStyleDeclaration

CSSCharsetRule

DOM Level 2 CSS

правило @charset в таблице стилей CSS **CSSRule → CSSCharsetRule**

Свойства

`String encoding`

Кодировка символов, заданная в правиле `@charset`. Если установить это свойство равным недопустимому значению, генерируется исключение DOMException с кодом, равным `SYNTAX_ERR`. Если правило или таблица стилей доступна только для чтения, то попытка установить значение этого свойства генерирует исключение DOMException с кодом `NO_ MODIFICATION_ALLOWED_ERR`.

Описание

Этот интерфейс представляет правило `@charset` таблицы стилей CSS. Подробности см. в литературе по CSS.

CSSFontFaceRule

DOM Level 2 CSS

правило @font-face в таблице стилей CSS **CSSRule → CSSFontFaceRule**

Свойства

`readonly CSSStyleDeclaration style`

Набор стилей для этого правила.

Описание

Этот интерфейс представляет правило `@font-face` для таблиц стилей CSS. Подробности см. в литературе по CSS.

CSSImportRule

DOM Level 2 CSS

правило @import в таблице стилей CSS **CSSRule → CSSImportRule**

Свойства

`readonly String href`

URL импортируемой таблицы стилей. Значение этого свойства не включает ограничитель «url()» вокруг значения.

`readonly MediaList media`

Список типов носителей, к которым применяется импортируемая таблица стилей.

`readonly CSSStyleSheet styleSheet`

Объект CSSStyleSheet, представляющий импортируемую таблицу стилей либо `null`, если таблица стилей до сих пор не загружена либо не была загружена, например, по причине неприменимости указанного типа носителя.

Описание

Этот интерфейс представляет правило `@import` таблицы стилей CSS. Свойство `styleSheet` представляет импортируемую таблицу стилей.

CSSMediaRule

DOM Level 2 CSS

правило @media в таблице стилей CSS **CSSRule → CSSMediaRule**

Свойства

`readonly CSSRuleList cssRules`

Массив (формально CSSRuleList) правил, вложенных в данный блок правил `@media`.

`readonly MediaList media`

Список типов носителей, к которым применяются вложенные правила.

Методы

`deleteRule()`

Удаляет вложенное правило в указанной позиции.

`insertRule()`

Вставляет новое правило в указанной позиции в данном блоке правил `@media`.

Описание

Этот интерфейс представляет правило `@media` и все его вложенные правила в таблице стилей CSS. Он определяет методы, позволяющие вставлять и удалять вложенные правила. Подробности см. в литературе по CSS.

CSSMediaRule.deleteRule() DOM Level 2 CSS

удаляет правило из блока @media

Синтаксис

```
void deleteRule(unsigned long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция удаляемого правила в блоке правил `@media`.

Исключения

Этот метод генерирует исключение DOMException с одним из перечисленных ниже значений `code` при следующих обстоятельствах:

`INDEX_SIZE_ERR`

индекс отрицателен или больше или равен количеству правил в `cssRules`.

`NO_MODIFICATION_ALLOWED_ERR`

Правило доступно только для чтения.

Описание

Этот метод удаляет правило в указанной позиции в массиве `cssRules`.

CSSMediaRule.insertRule() DOM Level 2 CSS

вставляет новое правило в блок @media

Синтаксис

```
unsigned long insertRule(String правило,
                         unsigned long индекс)
    throws DOMException;
```

Аргументы

правило

Полное представление добавляемого правила CSS в виде строки, которая может быть разобрана.

индекс

Позиция, в которую новое правило должно быть вставлено в массив `cssRules`, или `cssRules.length` для добавления нового правила в конец массива.

Возвращаемое значение

Значение аргумента *индекс*.

Исключения

Этот метод генерирует исключение DOMException с одним из перечисленных далее значений code при следующих обстоятельствах:

HIERARCHY_REQUEST_ERR

CSS не допускает наличие указанного *правила* в заданной позиции.

INDEX_SIZE_ERR

индекс отрицателен или больше, чем cssRules.length.

NO_MODIFICATION_ALLOWED_ERR

Правило @media и его массив cssRules доступны только для чтения.

SYNTAX_ERR

Указанное *правило* содержит синтаксическую ошибку.

Описание

Этот метод вставляет указанное *правило* в массив cssRules по указанному *индексу*.

CSSPageRule

DOM Level 2 CSS

правило @page в таблице стилей CSS **CSSRule → CSSPageRule**

Свойства

String selectorText

Текст селектора страницы для данного правила. Установка этого свойства равным недопустимому значению генерирует исключение DOMException с кодом SYNTAX_ERR. Установка этого свойства, когда правило доступно только для чтения, генерирует исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR.

readonly CSSStyleDeclaration style

Набор стилей для этого правила.

Описание

Этот интерфейс представляет правило @page в таблице стилей, которое обычно используется для задания формата страницы для печати. Подробности см. в литературе по CSS.

CSSPrimitiveValue

DOM Level 2 CSS

одиночное значение стиля CSS **CSSValue → CSSPrimitiveValue**

Константы

Следующие константы являются допустимыми значениями для свойства primitiveType. Они задают тип значения, а для числовых значений – единицы измерения, в которых это значение представлено.

unsigned short CSS_UNKNOWN = 0

Значение не распознано, и в данной реализации неизвестно, как его анализировать. Текстовое представление значения доступно через свойство cssText.

`unsigned short CSS_NUMBER = 1`

Безразмерное число. Возвращается методом `getFloatValue()`.

`unsigned short CSS_PERCENTAGE = 2`

Проценты. Возвращается методом `getFloatValue()`.

`unsigned short CSS_EMS = 3`

Относительный размер, измеренный в ems (высота текущего шрифта). Возвращается методом `getFloatValue()`.

`unsigned short CSS_EXS = 4`

Относительный размер, измеренный в exs («x-высота» текущего шрифта). Возвращается методом `getFloatValue()`.

`unsigned short CSS_PX = 5`

Размер, измеряемый в пикселах. Возвращается методом `getFloatValue()`. Размер в пикселах – это относительное значение, в том смысле, что оно зависит от разрешения дисплея и не может быть преобразовано в дюймы, миллиметры, пункты или другие абсолютные единицы. Однако пикселы – это также наиболее часто используемые единицы, и они рассматриваются как абсолютные, например для метода `AbstractView.getComputedStyle()`.

`unsigned short CSS_CM = 6`

Абсолютный размер в сантиметрах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_MM = 7`

Абсолютный размер в миллиметрах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_IN = 8`

Абсолютный размер в дюймах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_PT = 9`

Абсолютный размер в пунктах (1/72 часть дюйма). Возвращается методом `getFloatValue()`.

`unsigned short CSS_PC = 10`

Абсолютный размер в пиках (12 пунктов). Возвращается методом `getFloatValue()`.

`unsigned short CSS_DEG = 11`

Угол в градусах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_RAD = 12`

Угол в радианах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_GRAD = 13`

Угол в градах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_MS = 14`

Время в миллисекундах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_S = 15`

Время в секундах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_HZ = 16`

Частота в герцах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_KHZ = 17`

Частота в килогерцах. Возвращается методом `getFloatValue()`.

`unsigned short CSS_DIMENSION = 18`

Безразмерная величина. Возвращается методом `getFloatValue()`.

`unsigned short CSS_STRING = 19`

Строка. Возвращается методом `getStringValue()`.

`unsigned short CSS_URI = 20`

URI. Возвращается методом `getStringValue()`.

`unsigned short CSS_IDENT = 21`

Идентификатор. Возвращается методом `getStringValue()`.

`unsigned short CSS_ATTR = 22`

Функция-атрибут. Возвращается методом `getStringValue()`.

`unsigned short CSS_COUNTER = 23`

Счетчик. Возвращается методом `getCounterValue()`.

`unsigned short CSS_RECT = 24`

Прямоугольник. Возвращается методом `getRectValue()`.

`unsigned short CSS_RGBCOLOR = 25`

Цвет. Возвращается методом `getRGBColorValue()`.

Свойства

`readonly unsigned short primitiveType`

Тип данного значения. Это свойство содержит одну из констант, определенных в предыдущем разделе.

Методы

`getCounterValue()`

Для значений типа `CSS_COUNTER` возвращает объект Counter, представляющий это значение.

`getFloatValue()`

Возвращает числовое значение, преобразует его, если это необходимо, в указанные единицы измерения.

`getRectValue()`

Для значений типа `CSS_RECT` возвращает объект Rect, представляющий это значение.

`getRGBColorValue()`

Для значений типа `CSS_RGBCOLOR` возвращает объект RGBColor, представляющий значение.

`getStringValue()`

Возвращает значение в виде строки.

`setFloatValue()`

Устанавливает числовое значение равным указанному числу в указанных единицах измерения.

`setStringValue()`

Устанавливает строковое значение равным указанной строке указанного типа.

Описание

Подынтерфейс CSSValue представляет одиночное значение в CSS. Сравните с интерфейсом CSSValueList, представляющим список значений CSS. Слово «primitive» в названии этого интерфейса может ввести в заблуждение; этот интерфейс может представлять и некоторые сложные типы значений CSS, такие как счетчики, прямоугольники и цвета.

Свойство `primitiveType` содержит одну из ранее определенных констант и задает тип значения. Различные методы, определенные в этом интерфейсе, позволяют получать значения различных типов, а также устанавливать числовые и строковые значения.

См. также

Counter, CSSValue, CSSValueList, Rect, RGBColor

Тип: RGBColor.blue, RGBColor.green, RGBColor.red, Rect.bottom, Rect.left, Rect. right, Rect.top

CSSPrimitiveValue.getCounterValue() DOM Level 2 CSS

возвращает значение Counter

Синтаксис

```
Counter getCounterValue()
    throws DOMException;
```

Возвращаемое значение

Объект Counter, представляющий значение данного CSSPrimitiveValue.

Исключения

Этот метод генерирует исключение DOMException с кодом `INVALID_ACCESS_ERR`, если значение свойства `primitiveType` не равно `CSS_COUNTER`.

Описание

Этот метод возвращает объект Counter, представляющий CSS-счетчик. Соответствующий ему метод `setCounterValue()` отсутствует, но можно модифицировать значение путем установки свойств возвращаемого объекта Counter.

CSSPrimitiveValue.getFloatValue() DOM Level 2 CSS

получение числового значения, возможно, с преобразованием единиц измерения

Синтаксис

```
float getFloatValue(unsigned short unitType)
    throws DOMException;
```

Аргументы

unitType

Одна из констант типа CSSPrimitiveValue, задающая единицы измерения возвращаемого значения.

Возвращаемое значение

Числовое значение с плавающей точкой данного CSSPrimitiveValue, выраженное в указанных единицах измерения.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `INVALID_ACCESS_ERR`, если в данном объекте CSSPrimitiveValue содержится нечисловое значение или если значение не может быть преобразовано в запрашиваемые единицы измерения. (О преобразовании типов рассказано в следующем разделе.)

Описание

Метод преобразует числовые значения объектов CSSPrimitiveValue в указанные единицы и возвращает преобразованные значения.

Допускаются только определенные типы преобразований единиц измерения. Размеры могут быть преобразованы в размеры, углы – в углы, значения времени – в значения времени, а частоты – в частоты. Очевидно, что размер, выраженный в миллиметрах, не может быть преобразован в частоту, выраженную в килогерцах. Кроме того, не все размеры могут быть преобразованы. Относительные размеры (выраженные в ems, exs или пикселах) могут быть преобразованы в другие относительные размеры, но не могут быть преобразованы в абсолютные. Аналогично, абсолютные размеры нельзя преобразовать в относительные. И наконец, процентные значения нельзя преобразовать ни в какие другие единицы измерения, за исключением цветовых процентных значений, которые обозначают процент от 255 и могут быть преобразованы в тип `CSS_NUMBER`.

CSSPrimitiveValue.getRectValue() DOM Level 2 CSS

возвращает значение Rect

Синтаксис

```
Rect getRectValue()
    throws DOMException;
```

Возвращаемое значение

Объект Rect, представляющий значение данного CSSPrimitiveValue.

Исключения

Этот метод генерирует исключение DOMException с кодом `INVALID_ACCESS_ERR`, если значение свойства `primitiveType` не равно `CSS_RECT`.

Описание

Этот метод возвращает объект Rect, представляющий прямоугольник CSS. Соответствующий ему метод `setRectValue()` отсутствует, но можно модифицировать значение, устанавливая свойства возвращаемого объекта Rect.

CSSPrimitiveValue.getRGBColorValue() DOM Level 2 CSS

получает значение RGBColor

Синтаксис

```
RGBColor getRGBColorValue()
    throws DOMException;
```

Возвращаемое значение

Объект RGBColor, представляющий значение данного CSSPrimitiveValue.

Исключения

Этот метод генерирует исключение DOMException с кодом `INVALID_ACCESS_ERR`, если значение свойства `primitiveType` не равно `CSS_RGBCOLOR`.

Описание

Этот метод возвращает объект RGBColor, представляющий цвет. Соответствующий ему метод `setRGBColorValue()` отсутствует, но можно модифицировать значение, устанавливая свойства возвращаемого объекта RGBColor.

CSSPrimitiveValue.getStringValue() DOM Level 2 CSS

запрашивает строковое значение CSS

Синтаксис

```
String getStringValue()
    throws DOMException;
```

Возвращаемое значение

Строковое значение данного CSSPrimitiveValue.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `INVALID_ACCESS_ERR`, если свойство `primitiveType` не равно `CSS_STRING`, `CSS_URI`, `CSS_IDENT` или `CSS_ATTR`.

CSSPrimitiveValue.setFloatValue() DOM Level 2 CSS

устанавливает числовое значение

Синтаксис

```
void setFloatValue(unsigned short unitType,
                   float floatValue)
    throws DOMException;
```

Аргументы

unitType

Одна из констант CSSPrimitiveValue, задающая числовые единицы измерения для данного значения.

floatValue

Новое значение (в единицах *unitType*).

Исключения

Этот метод генерирует исключение DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если атрибут CSS, с которым связано это значение, доступен только для чтения. Он генерирует DOMException с кодом `INVALID_ACCESS_ERR`, если атрибут CSS не допускает числовых значений или не допускает значений с единицами измерения, обозначенными аргументом *unitType*.

Описание

Этот метод задает единицы измерения и числовое значение для данного объекта CSSPrimitiveValue.

CSSPrimitiveValue.setStringValue() DOM Level 2 CSS

устанавливает строковое значение

Синтаксис

```
void setStringValue(unsigned short stringType,
                    String stringValue)
    throws DOMException;
```

Аргументы

stringType

Тип устанавливаемой строки. Должен быть равен одной из констант CSSPrimitiveValue: CSS_STRING, CSS_URI, CSS_IDENT или CSS_ATTR.

stringValue

Новое устанавливаемое строковое значение.

Исключения

Этот метод генерирует исключение DOMException с кодом NO_MODIFICATION_ALLOWED_ERR, если атрибут CSS, с которым связано это значение, доступен только для чтения. Он генерирует DOMException с кодом INVALID_ACCESS_ERR, если атрибут CSS не допускает строковых значений или не допускает значений с типом, указанным в аргументе *stringType*.

Описание

Этот метод устанавливает строковое значение и строковый тип данного CSSPrimitiveValue.

CSSRule DOM Level 2 CSS

правило в таблице стилей CSS

Подынтерфейсы

CSSCharsetRule, CSSFontFaceRule, CSSImportRule, CSSMediaRule, CSSPageRule, CSSStyleRule, CSSUnknownRule

Константы

Эти константы представляют различные типы правил, которые могут присутствовать в таблице стилей CSS. Они являются допустимыми значениями свойства type и определяют, какой из приведенных выше подынтерфейсов реализует данный объект.

```
unsigned short UNKNOWN_RULE = 0;    // CSSUnknownRule
unsigned short STYLE_RULE = 1;      // CSSStyleRule
unsigned short CHARSET_RULE = 2;    // CSSCharsetRule
unsigned short IMPORT_RULE = 3;     // CSSImportRule
unsigned short MEDIA_RULE = 4;      // CSSMediaRule
unsigned short FONT_FACE_RULE = 5;  // CSSFontFaceRule
unsigned short PAGE_RULE = 6;       // CSSPageRule
```

Свойства

String cssText

Текстовое представление правила. Если установить это свойство, может возникнуть исключение DOMException (значения code и причины см. ниже):

`HIERARCHY_REQUEST_ERR`

Указанное правило недопустимо в данном месте таблицы стилей.

`INVALID_MODIFICATION_ERR`

Новое значение имеет тип, отличный от типа первоначального значения.

`NO_MODIFICATION_ALLOWED_ERR`

Правило или таблица стилей, содержащая это правило, доступны только для чтения.

`SYNTAX_ERR`

В указанной строке нарушен синтаксис CSS.

`readonly CSSRule parentRule`

Родительское правило для данного правила или значение `null`, если это правило не имеет родительского правила. Пример правила CSS, имеющего родительское правило, – это стилевое правило с правилом `@media`.

`readonly CSSStyleSheet parentStyleSheet`

Объект CSSStyleSheet, содержащий данное правило.

`readonly unsigned short type`

Тип правила CSS, представляемого данным объектом. Допустимые значения этого свойства представлены перечисленными выше константами. Интерфейс CSSRule никогда не реализуется непосредственно, и значение этого свойства указывает, какой более специфический подынтерфейс реализуется данным объектом.

Описание

Этот интерфейс определяет свойства, общие для всех типов правил в таблице стилей CSS. Ни один объект не реализует этот интерфейс непосредственно; вместо этого они реализуют один из более специфических подынтерфейсов, перечисленных выше. Наиболее важный подынтерфейс – это, пожалуй, CSSStyleRule, описывающий правило CSS, которое определяет стиль документа.

См. также

CSSStyleRule

Тип: CSSRule.parentRule, CSSStyleDeclaration.parentRule, CSSStyleSheet.ownerRule

Возвращается из: CSSRuleList.item()

CSSRuleList

DOM Level 2 CSS

массив объектов CSSRule

Свойства

`readonly unsigned long length`

Количество объектов CSSRule в данном массиве CSSRuleList.

Методы

`item()`

Возвращает объект CSSRule в указанной позиции. Вместо явного вызова этого метода JavaScript позволяет просто рассматривать объект CSSRuleList как массив и

обращаться к нему с помощью стандартной нотации массива с квадратными скобками. Если указанный индекс слишком велик, этот метод возвращает `null`.

Описание

Данный интерфейс определяет доступный только для чтения упорядоченный список (то есть массив) объектов CSSRule. Свойство `length` задает количество правил в списке, а метод `item()` позволяет извлекать правила в указанной позиции. В JavaScript объекты CSSRuleList ведут себя так же, как массивы JavaScript, и можно получить элемент из списка, используя нотацию массивов вместо вызова метода `item()`. (Заметьте, однако, что нотация массива не позволяет присваивать массиву CSSRuleList новые узлы.)

См. также

Тип: CSSMediaRule.cssRules, CSSStyleSheet.cssRules

CSSRuleList.item() DOM Level 2 CSS

получает CSSRule в указанной позиции

Синтаксис

`CSSRule item(unsigned long` *`индекс`*`);`

Аргументы

`индекс`

Позиция извлекаемого правила.

Возвращаемое значение

Объект CSSRule в указанной позиции или `null`, если *`индекс`* не является допустимой позицией.

CSSStyleDeclaration DOM Level 2 CSS

набор правил стиля CSS и их значений

Также реализует

Если реализация поддерживает модуль «CSS2» в дополнение к модулю «CSS» (как большинство веб-броузеров), то все объекты, реализующие этот интерфейс, также реализуют интерфейс CSS2Properties. CSS2Properties предоставляет часто используемые свойства быстрого доступа для установки и получения значений атрибутов CSS. Подробности см. в статье «CSS2Properties».

Свойства

`String cssText`

Текстовое представление атрибутов стилей и их значений. Значение этого свойства представляет собой полный текст стилевого правила, без селектора элемента и фигурных скобок, окружающих атрибуты и их значения. Присваивание этому свойству недопустимого значения приводит к возникновению исключения DOMException с кодом `SYNTAX_ERR`. Установка его для таблицы стилей или правила, доступных только для чтения, приводит к генерации DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`.

`readonly unsigned long length`

Количество атрибутов стиля в данном объявлении стиля.

`readonly CSSRule parentRule`

Объект CSSRule, содержащий данный CSSStyleDeclaration, или `null`, если данное объявление стиля не является частью правила CSS (как, например, для объектов CSSStyleDeclaration, представляющих встроенные атрибуты HTML `style`).

Методы

`getPropertyCSSValue()`

Возвращает объект CSSValue, представляющий значение атрибута CSS с указанным именем, или `null`, если данный атрибут не был явно установлен в данном блоке объявлений или если указанный стиль является атрибутом-сокращением.

`getPropertyPriority()`

Возвращает строку «important», если указанный атрибут CSS явно установлен в блоке объявления и для него указан признак приоритетности `!important`. Если атрибут не указан или не имеет приоритета, метод возвращает пустую строку.

`getPropertyValue()`

Возвращает значение указанного атрибута CSS в виде строки. Возвращает пустую строку, если атрибут не указан в данном блоке объявлений.

`item()`

Возвращает имя атрибута CSS в указанной позиции в данном блоке объявлений стиля. В JavaScript объект CSSStyleDeclaration может рассматриваться как массив и индексироваться с помощью квадратных скобок, а не посредством данного свойства. См. описание свойства `length`.

`removeProperty()`

Удаляет атрибут CSS с указанным именем из указанного блока объявлений.

`setProperty()`

Устанавливает значение указанного атрибута CSS равным указанному строковому значению, а также приоритет для данного блока объявлений.

Описание

Данный атрибут представляет блок объявления стиля в CSS: набор атрибутов CSS и их значений, разделенных точками с запятой. Блок объявления стиля является частью стилевого правила, заключенного в таблице стилей CSS в фигурные скобки. Значение атрибута HTML `style` также образует блок объявления стиля.

Метод `item()` и свойство `length` позволяют осуществлять цикл по именам всех CSS-атрибутов, указанных в данном блоке объявлений. В JavaScript также можно рассматривать объект CSSStyleDeclaration как массив и индексировать его с помощью нотации квадратных скобок, обходясь без явного вызова метода `item()`. Если в данном объявлении указаны имена атрибутов CSS, то для получения значений этих атрибутов можно прибегнуть к другим методам данного интерфейса. `getPropertyValue()` возвращает значение в виде строки, а `getPropertyCSSValue()` возвращает значение атрибута в виде объекта CSSValue. (Обратите внимание, что в DOM API атрибуты стиля CSS называются «свойствами» (properties). Здесь я потребляю термин «атрибуты» (attributes), чтобы не путать их со свойствами объектов JavaScript.)

В большинстве веб-броузеров каждый объект, реализующий интерфейс CSSStyleDeclaration, также реализует интерфейс CSS2Properties, который определяет свойство для каждого CSS-атрибута, определенного в спецификации CSS2. Значения этих свойств можно получать и устанавливать посредством методов `getPropertyValue()` и `setProperty()`.

См. также

CSS2Properties

Тип: CSSFontFaceRule.style, CSSPageRule.style, CSSStyleRule.style, HTMLElement. style

Возвращается из: Document.getOverrideStyle(), AbstractView.getComputedStyle()

CSSStyleDeclaration.getPropertyCSSValue() DOM Level 2 CSS

возвращает значение CSS-атрибута в виде объекта

Синтаксис

```
CSSValue getPropertyCSSValue(String имяСвойства);
```

Аргументы

имяСвойства

Имя требуемого атрибута CSS.

Возвращаемое значение

Объект CSSValue, представляющий значение атрибута с указанным именем, если он явно указан в данном объявлении стиля, или `null`, если данный атрибут не указан. Этот метод также возвращает `null`, если *имяСвойства* задает атрибут-сокращение CSS, так как атрибуты-сокращения содержат более одного значения и не могут быть представлены объектами CSSValue.

CSSStyleDeclaration.getPropertyPriority() DOM Level 2 CSS

получает приоритет атрибута CSS

Синтаксис

```
String getPropertyPriority(String имяСвойства);
```

Аргументы

имяСвойства

Имя атрибута CSS.

Возвращаемое значение

Строка «important», если данный атрибут CSS явно указан в данном блоке объявлений и имеет модификатор приоритета `!important`. Иначе возвращает пустую строку.

CSSStyleDeclaration.getPropertyValue() DOM Level 2 CSS

получает значение атрибута CSS в виде строки

Синтаксис

```
String getPropertyValue(String имяСвойства);
```

Аргументы

имяСвойства

Имя атрибута CSS, значение которого требуется получить.

Возвращаемое значение

Строковое значение атрибута CSS с указанным именем или пустая строка, если атрибут не установлен явно в данном блоке объявлений.

Описание

Этот метод возвращает значение указанного атрибута CSS в виде строки. В отличие от `getPropertyCSSValue()`, этот метод работает с атрибутами-сокращениями так же, как с обычными атрибутами. См. также описания других вспомогательных свойств интерфейса CSS2Properties.

CSSStyleDeclaration.item() DOM Level 2 CSS

получает имя атрибута CSS в указанной позиции

Синтаксис

```
String item(unsigned long индекс);
```

Аргументы

индекс

Позиция атрибута CSS, имя которого требуется получить.

Возвращаемое значение

Имя атрибута CSS в позиции *индекс* или пустая строка, если *индекс* отрицателен или больше или равен свойству `length`.

Описание

Интерфейс CSSStyleDeclaration представляет коллекцию атрибутов стиля CSS и их значений. Этот метод позволяет получить имя атрибута CSS по его позиции и в сочетании со свойством `length` позволяет выполнять цикл по набору атрибутов CSS, указанных в данном стилевом объявлении. Обратите внимание, что порядок атрибутов CSS, возвращаемых данным методом, не всегда совпадает с порядком, в котором они присутствуют в исходном коде документа или таблицы стилей.

В качестве альтернативы данному методу `item()` JavaScript позволяет рассматривать объект CSSStyleDeclaration как массив имен атрибутов CSS и использовать стандартный синтаксис работы с массивами для получения имени атрибута в указанной позиции.

CSSStyleDeclaration.removeProperty() DOM Level 2 CSS

удаляет спецификацию атрибута CSS

Синтаксис

```
String removeProperty(String имяСвойства)
    throws DOMException;
```

Аргументы

имяСвойства

Имя атрибута CSS, который должен быть удален.

Возвращаемое значение

Значение атрибута CSS с указанным именем в виде строки или пустая строка, если атрибут с указанным именем не был явно указан в данном объявлении стиля.

Исключения

Если данное объявление стиля доступно только для чтения, этот метод генерирует исключение с кодом `NO_MODIFICATION_ALLOWED_ERR`.

Описание

Этот метод удаляет указанный атрибут из блока атрибутов стиля и возвращает значение атрибута.

CSSStyleDeclaration.setProperty() DOM Level 2 CSS

устанавливает атрибут стиля CSS

Синтаксис

```
void setProperty(String имяСвойства,
                 String значение,
                 String свойство)
    throws DOMException;
```

Аргументы

имяСвойства

Имя устанавливаемого атрибута CSS.

значение

Новое значение атрибута в виде строки.

свойство

Новый приоритет атрибута. Этот аргумент необходимо установить равным «important», если спецификация атрибута должна быть отмечена как приоритетная (`!important`); в противном случае аргументу следует присвоить пустую строку.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `SYNTAX_ERR`, если указанное *значение* имеет неверный формат. Если для объявления стиля или атрибута был установлен доступ только на чтение, метод генерирует DOMException с кодом, равным `NO_MODIFICATION_ ALLOWED_ERR`.

Описание

Этот метод добавляет атрибут CSS с указанным именем, его значение и приоритет в стилевое объявление или, если объявление уже содержит значение для атрибута с указанным именем, устанавливает значение и приоритет для существующего атрибута.

Применение `setProperty()` для добавления нового атрибута CSS в объявление стиля может привести к вставке нового атрибута в любую позицию и фактически полностью изменить порядок всех существующих атрибутов. Следовательно, лучше не использовать `setProperty()` при обходе имен атрибутов с помощью метода `item()`.

CSSStyleRule

DOM Level 2 CSS

правило стиля в таблице стилей CSS **CSSRule → CSSStyleRule**

Свойства

`String selectorText`

Текст селектора, определяющий элементы документа, к которым применяется это правило. Установка этого свойства генерирует исключение DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если правило доступно только для чтения, или с кодом `SYNTAX_ERR`, если новое значение не соответствует синтаксическим правилам CSS.

`readonly CSSStyleDeclaration style`

Стилевые значения, которые должны применяться к элементам, определяемым с помощью `selectorText`.

Описание

Этот интерфейс представляет стилевое правило в таблице стилей CSS. Стилевые правила – это наиболее распространенный и важный тип правил в таблицах стилей: они содержат информацию о стилях, которая должна применяться к указанному набору элементов документа. `selectorText` – это строковое представление селектора элементов для этого правила, а `style` – это объект CSSStyleDeclaration, представляющий набор имен и значений стилей, применяемых к выбранным элементам.

См. также

CSSStyleDeclaration

CSSStyleSheet

DOM Level 2 CSS

таблица стилей CSS **StyleSheet → CSSStyleSheet**

Свойства

`readonly CSSRuleList cssRules`

Массив (формально CSSRuleList) объектов CSSRule, составляющих стилевую таблицу. Кроме фактических стилевых правил он включает все @-правила.

`readonly CSSRule ownerRule`

Если таблица стилей была импортирована с помощью правила `@import` в другую таблицу, это свойство содержит объект CSSImportRule, представляющий правило `@import`. В противном случае оно равно `null`. Если это свойство не равно `null`, то унаследованное свойство `ownerNode` равно `null`.

Методы

`deleteRule()`

Удаляет правило в указанной позиции.

`insertRule()`

Вставляет новое правило в указанную позицию.

Описание

Этот интерфейс представляет таблицу стилей CSS. В свойстве `cssRules` перечислены правила, содержащиеся в таблице стилей, а методы `insertRule()` и `deleteRule()` позволяют добавлять и удалять правила из этого списка.

См. также

StyleSheet

Тип: CSSImportRule.styleSheet, CSSRule.parentStyleSheet

Возвращается из: DOMImplementation.createCSSStyleSheet()

CSSStyleSheet.deleteRule() DOM Level 2 CSS

удаляет правила из таблицы стилей

Синтаксис

```
void deleteRule(unsigned long индекс)
    throws DOMException;
```

Аргументы

индекс

Индекс правила в массиве cssRules, которое должно быть удалено.

Исключения

Этот метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если *индекс* отрицателен или больше или равен cssRules.length. Если таблица стилей доступна только для чтения, он генерирует DOMException с кодом NO_MODIFICATION_ALLOWED_ERR.

Описание

Метод удаляет правило в позиции, указанной аргументом *индекс*, из массива cssRules.

CSSStyleSheet.insertRule() DOM Level 2 CSS

вставляет правило в таблицу стилей

Синтаксис

```
unsigned long insertRule(String правило,
                         unsigned long индекс)
    throws DOMException;
```

Аргументы

правило

Полное, подходящее для разбора текстовое представление правила, которое должно быть добавлено в таблицу стилей. Для стилевых правил оно включает как селектор элемента, так и информацию о стиле.

индекс

Позиция в массиве cssRules, где должно быть вставлено или добавлено правило.

Возвращаемое значение

Значение аргумента *индекс*.

Исключения

Этот метод генерирует исключение DOMException с одним из перечисленных ниже значений code в следующих ситуациях:

HIERARCHY_REQUEST_ERR

Синтаксис CSS не допускает появления указанного правила в данном месте.

`INDEX_SIZE_ERR`

индекс отрицателен или больше, чем `cssRules.length`.

`NO_MODIFICATION_ALLOWED_ERR`

Таблица стилей доступна только для чтения.

`SYNTAX_ERR`

Указанный текст *правила* содержит синтаксическую ошибку.

Описание

Этот метод вставляет (или добавляет) новое *правило* CSS в позиции, указанной аргументом *индекс* в массиве `cssRules` данной таблицы стилей.

CSSUnknownRule

DOM Level 2 CSS

нераспознаваемое правило в таблице стилей CSS **CSSRule → CSSUnknownRule**

Описание

Этот интерфейс представляет правило в таблице стилей CSS, которое не было распознано броузером и не могло быть разобрано (обычно по причине того, что определено в версии стандарта CSS, которая не поддерживается броузером). Обратите внимание, что этот интерфейс не определяет никаких собственных свойств или методов. Текст нераспознаваемого правила доступен через унаследованное свойство `cssText`.

CSSValue

DOM Level 2 CSS

значение атрибута стиля CSS

Подынтерфейсы

CSSPrimitiveValue, CSSValueList

Константы

Следующие константы задают допустимые значения для свойства `cssValueType`:

`unsigned short CSS_INHERIT = 0`

Эта константа представляет специальное значение «inherit», обозначающее, что фактическое значение атрибута стиля CSS наследуется. Свойство `cssText` в этом случае равно «inherit».

`unsigned short CSS_PRIMITIVE_VALUE = 1`

Значение относится к элементарному типу. Этот объект CSSValue также реализует более специфический подынтерфейс CSSPrimitiveValue.

`unsigned short CSS_VALUE_LIST = 2`

Значение является составным, состоящим из списка значений. Этот объект CSSValue также реализует более специфический подынтерфейс CSSValueList и ведет себя как массив объектов CSSValue.

`unsigned short CSS_CUSTOM = 3`

Эта константа определена, чтобы допускать расширения объектной модели CSS. Она обозначает, что данное значение CSSValue представляет значение некоторого типа, не определенного стандартом CSS или DOM. Если вы работаете с реализацией, поддерживающей такие расширения, то объект CSSValue может также реали-

зовывать какой-то другой интерфейс (например, интерфейс SVGColor, определенный в стандарте Scalable Vector Graphics), который можно использовать.

Свойства

`String cssText`

Текстовое представление значения. Установка этого свойства может генерировать исключение DOMException. Код `SYNTAX_ERR` обозначает, что новое значение не соответствует синтаксису CSS. Код `INVALID_MODIFICATION_ERR` обозначает, что была предпринята попытка установить значение, тип которого отличен от типа исходного значения. Код `NO_MODIFICATION_ALLOWED_ERR` обозначает, что значение доступно только для чтения.

`readonly unsigned short cssValueType`

Тип представляемого этим объектом значения. Четыре допустимых значения этого свойства определяются перечисленными выше константами.

Описание

Этот интерфейс представляет значение атрибута CSS. Свойство `cssText` содержит это значение в текстовом формате. Если свойство `cssValueType` равно `CSSValue.CSS_PRIMITIVE_VALUE`, то этот объект CSSValue также реализует более специфический интерфейс CSSPrimitiveValue. Если `cssValueType` равен `CSSValue.CSS_VALUE_LIST`, то данное значение CSSValue представляет список значений и реализует интерфейс CSSValueList.

См. также

CSSPrimitiveValue, CSSValueList

Возвращается из: CSSStyleDeclaration.getPropertyCSSValue(), CSSValueList.item()

CSSValueList

DOM Level 2 CSS

CSSValue, содержащий массив объетов CSSValue **CSSValue → CSSValueList**

Свойства

`readonly unsigned long length`

Число объектов CSSValue в данном массиве.

Методы

`item()`

Возвращает объект CSSValue в указанной позиции в массиве или `null`, если указанная позиция отрицательна или больше или равна `length`.

Описание

Этот интерфейс представляет массив объектов CSSValue и сам является значением типа CSSValue. Для извлечения объекта CSSValue из указанной позиции может применяться метод `item()`, но в JavaScript проще работать с массивом с помощью стандартной нотации массивов с квадратными скобками.

Порядок объектов CSSValue в массиве CSSValueList соответствует порядку их присутствия в стилевом объявлении CSS. Некоторые атрибуты CSS, значения которых являются объектами CSSValueList, также могут иметь значение `none`. Это специальное значение транслируется в объект CSSValueList со значением `length`, равным 0.

CSSValueList.item() DOM Level 2 CSS

получает CSSValue в указанной позиции

Синтаксис

```
CSSValue item(unsigned long индекс);
```

Аргументы

индекс

Позиция требуемого значения CSSValue.

Возвращаемое значение

Объект CSSValue в указанной позиции в данном CSSValueList или `null`, если *индекс* отрицателен или больше или равен `length`.

Document DOM Level 1 Core

HTML- или XML-документ **Node → Document**

Подынтерфейсы

HTMLDocument

Также реализует

`DocumentCSS`

Если реализация поддерживает модуль CSS, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentCSS и его метод `getOverrideStyle()`.

`DocumentEvent`

Если реализация поддерживает модуль Events, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentEvent и его метод `createEvent()`.

`DocumentRange`

Если реализация поддерживает модуль Range, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentRange и его метод `createRange()`.

`DocumentStyle`

Если реализация поддерживает модуль StyleSheets, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentStyle и его свойство `styleSheets`.

`DocumentTraversal`

Если реализация поддерживает модуль Traversal, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentTraversal и его методы `createNodeIterator()` и `createTreeWalker()`.

`DocumentView`

Если реализация поддерживает модуль Views, то объект, реализующий данный интерфейс Document, также реализует интерфейс DocumentView и его свойство `defaultView`.

Поскольку эти интерфейсы определяют часто реализуемые дополнения к интерфейсу Document, их свойства и методы перечислены и описаны здесь, как если бы они были непосредственной частью интерфейса Document.

Свойства

`readonly AbstractView defaultView` *[DOM Level 2 Views]*

Представление документа по умолчанию. В среде веб-броузера это свойство задает объект Window (реализующий интерфейс AbstractView), в котором отображается документ.

Обратите внимание, что формально это свойство является частью интерфейса DocumentView; оно определяется объектом Document только в реализациях, поддерживающих модуль Views.

`readonly DocumentType doctype`

Для XML-документов с декларацией `<!DOCTYPE>` задает узел DocumentType, представляющий DTD документа. Для HTML-документов и XML-документов без `<!DOCTYPE>` это свойство равно `null`. Это свойство доступно только для чтения, и узел, на который оно ссылается, тоже доступен только для чтения.

`readonly Element documentElement`

Ссылка на корневой элемент документа. Для HTML-документов это свойство всегда является объектом Element, представляющим тег `<html>`. Этот корневой элемент также доступен через массив `childNodes[]`, унаследованный от Node.

`readonly DOMImplementation implementation`

Объект DOMImplementation, представляющий реализацию, создавшую этот документ.

`readonly StyleSheetList styleSheets` *[DOM Level 2 StyleSheets]*

Коллекция объектов, представляющих все таблицы стилей, встроенные в документ, или на которые есть ссылки из него. В HTML-документах они включают таблицы стилей, определенные с помощью тегов `<link>` и `<style>`.

Обратите внимание, что это свойство формально является частью интерфейса DocumentStyle; оно определяется объектом Document только в реализациях, поддерживающих модуль StyleSheets.

Методы

`createAttribute()`

Создает новый узел Attr с указанным именем.

`createAttributeNS()` *[DOM Level 2]*

Создает новый узел Attr с указанным именем и пространством имен.

`createCDATASection()`

Создает новый узел CDATASection, содержащий указанный текст.

`createComment()`

Создает новый узел Comment, содержащий указанную строку.

`createDocumentFragment()`

Создает новый пустой узел DocumentFragment.

`createElement()`

Создает новый узел Element с указанным именем тега.

`createElementNS()` *[DOM Level 2]*

Создает новый узел Element с указанным именем и пространством имен.

`createEntityReference()`

Создает узел EntityReference, который ссылается на сущность с указанным именем. Если объект DocumentType для данного документа определяет сущность с данным именем, создаваемому узлу EntityReference даются те же доступные только для чтения дочерние узлы, что имеются в узле Entity.

`createEvent()` *[DOM Level 2 Events]*

Создает новый синтетический объект Event указанного типа. Формально этот метод определяется интерфейсом DocumentEvent; он реализуется в объекте Document только в реализациях, поддерживающих модуль Events.

`createNodeIterator()` *[DOM Level 2 Traversal]*

Создает объект NodeIterator. Этот метод формально является частью интерфейса DocumentTraversal; он реализуется в объекте Document только в реализациях, поддерживающих модуль Traversal.

`createProcessingInstruction()`

Создает новый узел ProcessingInstruction с указанной целью и строкой данных.

`createRange()` *[DOM Level 2 Range]*

Создает новый объект Range. Этот метод формально является частью интерфейса DocumentRange; он реализуется объектом Document только в реализациях, поддерживающих модуль Range.

`createTextNode()`

Создает новый узел Text, представляющий указанный текст.

`createTreeWalker()` *[DOM Level 2 Traversal]*

Создает объект TreeWalker. Этот метод формально является частью интерфейса DocumentTraversal; он реализуется объектом Document только в реализациях, поддерживающих модуль Traversal.

`getElementById()` *[DOM Level 2]*

Возвращает элемент-потомок данного документа, имеющий указанное значение своего атрибута `id`, или `null`, если такой элемент в документе отсутствует.

`getElementsByTagName()`

Возвращает массив (формально NodeList) всех узлов Element в данном документе, имена которых совпадают с указанным именем тега. Узлы Element расположены в результирующем массиве в том же порядке, что и в исходном тексте документа.

`getElementsByTagNameNS()` *[DOM Level 2]*

Возвращает массив всех узлов элемента, имеющих указанное имя тега и пространство имен.

`getOverrideStyle()` *[DOM Level 2 CSS]*

Получает данные о перекрывающем стиле CSS для указанного элемента (и, возможно, для псевдоэлемента с указанным именем). Этот метод формально является частью интерфейса DocumentCSS и реализуется объектом Document только в реализациях, поддерживающих модуль CSS.

`importNode()` *[DOM Level 2]*

Делает копию узла из некоторого другого документа, подходящую для вставки в этот документ.

Описание

Интерфейс Document – это корневой узел дерева документа. Document может иметь несколько дочерних узлов, но только один из них может быть узлом Element – это корневой элемент документа. К корневому элементу проще всего обращаться через свойство `documentElement`. Свойства `doctype` и `implementation` предоставляют доступ к объекту DocumentType (если он есть) и к объекту DOMImplementation данного документа.

Большинство методов, определенных в интерфейсе Document, – это «методы-фабрики», применяемые для создания различных типов узлов, которые могут быть вставлены в документ. К исключениям можно отнести методы `getElementById()` и `getElementsByTagName()`, особенно полезные для поиска определенного элемента или набора связанных элементов внутри дерева документа.

Этот объект Document можно сравнить с объектом Document, описанным в «Справочнике по клиентскому JavaScript» в этой книге. Свойства и методы Level 0 этого объекта Document формально определяются стандартом DOM в интерфейсе HTMLDocument. См. DOM-эквивалент традиционного объекта Document клиентского JavaScript в статье «HTMLDocument» в этом справочнике.

Интерфейс Document определен в модуле Core спецификации DOM Level 2. Несколько других модулей определяют интерфейсы-«дополнения», предназначенные для реализации тем же объектом, который реализует интерфейс Document. Например, если реализация поддерживает модуль CSS, объект, реализующий этот интерфейс, также реализует интерфейс DocumentCSS. В JavaScript свойства и методы этих дополнительных интерфейсов могут применяться так же, как если бы они были определены в Document, и по этой причине такие свойства и методы перечислены здесь. См. выше раздел «Также реализует», где приведен полный список интерфейсов-«дополнений» для Document.

См. также

HTMLDocument

Тип: AbstractView.document, HTMLFrameElement.contentDocument, HTMLIFrameElement.contentDocument, HTMLObjectElement.contentDocument, Node. ownerDocument

Возвращается из: DOMImplementation.createDocument()

Document.createAttribute() DOM Level 1 Core

создает новый узел Attr

Синтаксис

```
Attr createAttribute(String имя)
    throws DOMException;
```

Аргументы

имя

Имя создаваемого атрибута.

Возвращаемое значение

Только что созданный узел Attr со свойством nodeName, равным *имени*.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным INVALID_CHARACTER_ERR, если *имя* содержит недопустимые символы.

См. также

Attr, Element.setAttribute(), Element.setAttributeNode()

Document.createAttributeNS() DOM Level 2 Core

создает узел Attr с указанным именем и пространством имен

Синтаксис

```
Attr createAttributeNS(String URIпространстваИмен,
                       String уточненноеИмя)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для Attr или null в случае отсутствия пространства имен.

уточненноеИмя

Уточненное имя атрибута, которое должно включать префикс пространства имен, двоеточие и локальное имя.

Возвращаемое значение

Только что созданный узел Attr с указанным именем и пространством имен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных далее значений code в следующих ситуациях:

INVALID_CHARACTER_ERR

уточненноеИмя содержит недопустимый символ.

NAMESPACE_ERR

уточненноеИмя имеет неверный формат, или имеется несоответствие между *уточненнымИменем* и *URIпространстваИмен*.

NOT_SUPPORTED_ERR

Реализация не поддерживает XML-документов и, следовательно, в ней нет этого метода.

Описание

Метод createAttributeNS() идентичен методу createAttribute() за исключением того, что созданный узел Attr имеет имя и пространство имен, а не просто имя. Применение этого метода имеет смысл только в XML-документах, использующих пространства имен.

Document.createCDATASection() DOM Level 1 Core

создает новый узел CDATASection

Синтаксис

```
CDATASection createCDATASection(String данные)
    throws DOMException;
```

Аргументы

данные

Текст в создаваемом разделе CDATA.

Возвращаемое значение

Только что созданный узел CDATASection, содержимым которого являются указанные *данные*.

Исключения

Для HTML-документов этот метод генерирует исключение DOMException с кодом, равным `NOT_SUPPORTED_ERR`, так как в HTML-документах наличие узлов CDATASection не допускается.

Document.createComment() DOM Level 1 Core

создает новый узел Comment

Синтаксис

```
Comment createComment(String данные);
```

Аргументы

данные

Текст создаваемого узла Comment.

Возвращаемое значение

Только что созданный узел Comment, текстом которого являются указанные *данные*.

Document.createDocumentFragment() DOM Level 1 Core

создает новый пустой узел DocumentFragment

Синтаксис

```
DocumentFragment createDocumentFragment();
```

Возвращаемое значение

Только что созданный узел DocumentFragment, не содержащий дочерних узлов.

Document.createElement() DOM Level 1 Core

создает новый узел Element

Синтаксис

```
Element createElement(String имяТега)
    throws DOMException;
```

Аргументы

имяТега

Имя тега создаваемого элемента. Теги HTML безразличны к регистру, тогда как в именах тегов XML регистр букв имеет значение.

Возвращаемое значение

Только что созданный узел Element с указанным именем тега.

Исключения

Этот метод генерирует исключение DOMException с кодом `INVALID_CHARACTER_ERR`, если *имяТега* содержит недопустимый символ.

Document.createElementNS() DOM Level 2 Core

создает новый узел Element, используя пространство имен

Синтаксис

```
Element createElementNS(String URIпространстваИмен,
                        String уточненноеИмя)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Уникальный идентификатор пространства имен для нового элемента или `null` в случае отсутствия пространства имен.

уточненноеИмя

Уточненное имя нового узла Element. Должно включать префикс пространства имен, двоеточие и локальное имя.

Возвращаемое значение

Только что созданный узел Element с указанным именем тега и пространством имен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

уточненноеИмя содержит недопустимый символ.

`NAMESPACE_ERR`

Формат *уточненногоИмени* неверен или имеется несоответствие между *уточненнымИменем* и *URIпространстваИмен*.

`NOT_SUPPORTED_ERR`

Реализация не поддерживает XML-документы и, следовательно, в ней нет этого метода.

Описание

Метод `createElementNS()` идентичен методу `createElement()` за исключением того, что у созданного узла Element не просто есть имя, он именован и определен в пространстве имен. Применение этого метода имеет смысл только в XML-документах, использующих пространства имен.

Document.createEntityReference() DOM Level 1 Core

создает новый узел EntityReference

Синтаксис

```
EntityReference createEntityReference(String имя)
    throws DOMException;
```

Аргументы

имя

Имя сущности, на которую указывает ссылка.

Возвращаемое значение

Новый узел EntityReference, ссылающийся на сущность с указанным именем.

Исключения

Этот метод может генерировать исключение DOMException с одним из следующих значений `code`:

`INVALID_CHARACTER_ERR`

Указанное имя сущности содержит недопустимый символ.

`NOT_SUPPORTED_ERR`

Это HTML-документ, и он не поддерживает ссылки на сущности.

Описание

Этот метод создает и возвращает узел EntityReference, который ссылается на сущность с указанным именем. Обратите внимание, что он всегда генерирует исключение, если это HTML-документ, так как HTML не допускает ссылок на сущности. Если в документе есть DocumentType и если в этом DocumentType определяется объект Entity с указанным именем, то возвращаемый EntityReference имеет те же дочерние узлы, что и узел Entity, на который указывает ссылка.

Document.createEvent() DOM Level 2 Events

создает объект Event

Синтаксис

```
Event createEvent(String типСобытий)
    throws DOMException;
```

Аргументы

типСобытий

Имя модуля событий, для которого требуется создать объект Event. Список допустимых типов событий см. в разделе «Описание».

Возвращаемое значение

Только что созданный объект Event указанного типа.

Исключения

Этот метод генерирует исключение DOMException с кодом `NOT_SUPPORTED_ERR`, если реализация не поддерживает события запрашиваемого типа.

Описание

Этот метод создает новое событие, имеющее тип, указанный в аргументе *типСобытий*. Обратите внимание, что значение этого аргумента должно быть не именем создаваемого интерфейса событий (в единственном числе), а именем (во множественном числе) модуля DOM, определяющего этот интерфейс. В следующей таблице приведены допустимые значения *типаСобытий* и интерфейсы событий, создаваемые каждым из этих значений.

Аргумент eventType	Интерфейс событий	Метод инициализации
HTMLEvents	Event	`initEvent()`
MouseEvents	MouseEvent	`initMouseEvent()`
UIEvents	UIEvent	`initUIEvent()`
MutationEvents	MutationEvent	`initMutationEvent()`

Создав объект Event с помощью этого метода, необходимо инициализировать объект, вызвав указанный в таблице метод инициализации. Подробности о методе инициализации см. на справочной странице соответствующего интерфейса событий.

Этот метод фактически определяется не интерфейсом Document, а интерфейсом DocumentEvent. Если реализация поддерживает модуль Events, то объект Document всегда реализует интерфейс DocumentEvent и поддерживает этот метод.

См. также

Event, MouseEvent, MutationEvent, UIEvent

Document.createNodeIterator()

DOM Level 2 Traversal

создает NodeIterator для данного документа

Синтаксис

```
NodeIterator createNodeIterator(Node корневойЭлемент,
                                unsigned long чтоПоказывать,
                                NodeFilter фильтр,
                                boolean разворачиватьСсылкиНаСущности)
    throws DOMException;
```

Аргументы

корневойЭлемент

Корень поддерева, с которым объект NodeIterator должен проводить итерации.

чтоПоказывать

Битовая маска из одного или нескольких флагов NodeFilter, указывающих, какие типы узлов должен возвращать данный NodeIterator.

фильтр

Необязательная функция фильтрации узлов для NodeIterator или `null` в случае отсутствия фильтра узлов.

разворачиватьСсылкиНаСущности

`true`, если NodeIterator должен разворачивать ссылки на сущности в XML-документах, или `false` в противном случае.

Возвращаемое значение

Только что созданный объект NodeIterator.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `NOT_SUPPORTED_ERR`, если аргумент *корневойЭлемент* равен `null`.

Описание

Этот метод создает и возвращает новый объект NodeIterator для обхода поддерева, начинающегося с узла *корневойЭлемент*, с использованием указанных фильтров.

Этот метод фактически не является частью интерфейса Document, а относится к интерфейсу DocumentTraversal. Если реализация поддерживает модуль Traversal, объект Document всегда реализует DocumentTraversal и определяет этот метод.

См. также

Document.createTreeWalker(), NodeFilter, NodeIterator

Document.createProcessingInstruction() DOM Level 1 Core

создает узел ProcessingInstruction

Синтаксис

```
ProcessingInstruction createProcessingInstruction(String цель,
                                                  String данные)
    throws DOMException;
```

Аргументы

цель

Цель инструкции обработки.

данные

Текст содержимого инструкции обработки.

Возвращаемое значение

Только что созданный узел ProcessingInstruction.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

Указанная *цель* содержит недопустимый символ.

`NOT_SUPPORTED_ERR`

Это документ HTML, и инструкции обработки не поддерживаются.

Document.createRange() DOM Level 2 Range

создает объект Range

Синтаксис

```
Range createRange();
```

Возвращаемое значение

Только что созданный объект Range, в котором обе граничные точки установлены на начало документа.

Описание

Этот метод создает объект Range, который может использоваться для представления области данного документа или области из DocumentFragment, связанного с этим документом.

Обратите внимание, что этот метод фактически определен не в интерфейсе Document, а в интерфейсе DocumentRange. Если реализация поддерживает модуль Range, то объект Document всегда реализует DocumentRange и определяет этот метод.

Document.createTextNode() DOM Level 1 Core

создает новый узел Text

Синтаксис

```
Text createTextNode(String данные);
```

Аргументы

данные

Содержимое узла Text.

Возвращаемое значение

Только что созданный узел Text, представляющий указанную строку *данных*.

Document.createTreeWalker() DOM Level 2 Traversal

создает TreeWalker для данного документа

Синтаксис

```
TreeWalker createTreeWalker(Node корневойЭлемент,
                            unsigned long чтоПоказывать,
                            NodeFilter фильтр,
                            boolean разворачиватьСсылкиНаСущности)
    throws DOMException;
```

Аргументы

корневойЭлемент

Корень поддерева, которое должен обходить TreeWalker.

чтоПоказывать

Битовая маска из одного или нескольких флагов NodeFilter, указывающих, какие типы узлов должен возвращать данный TreeWalker.

фильтр

Необязательная функция фильтрации узлов для TreeWalker или `null` в случае отсутствия фильтра узлов.

разворачиватьСсылкиНаСущности

`true`, если TreeWalker должен разворачивать ссылки на сущности в XML-документах, или `false` в противном случае.

Возвращаемое значение

Только что созданный объект TreeWalker.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `NOT_SUPPORTED_ERR`, если аргумент *корневойЭлемент* равен `null`.

Описание

Этот метод создает и возвращает новый объект TreeWalker для обхода поддерева, начинающегося с узла *корневойЭлемент*, с использованием указанных фильтров.

Этот метод фактически не является частью интерфейса Document, а определен в интерфейсе DocumentTraversal. Если реализация поддерживает модуль Traversal, то объект Document всегда реализует DocumentTraversal и определяет этот метод.

См. также

Document.createNodeIterator(), NodeFilter, TreeWalker

Document.getElementById() DOM Level 2 Core; в DOM Level 1 определен в HTMLDocument

находит элемент с указанным уникальным ID

Синтаксис

```
Element getElementById(String IdЭлемента);
```

Аргументы

IdЭлемента

Значение атрибута `id` требуемого элемента.

Возвращаемое значение

Узел Element, представляющий элемент документа с указанным атрибутом `id`, или `null`, если такой элемент не найден.

Описание

Метод ищет в документе узел Element с атрибутом `id`, значение которого равно *IdЭлемента*, и возвращает этот элемент. Если такой элемент не найден, он возвращает `null`. Значение атрибута `id` предполагается уникальным в пределах документа, а если этот метод находит более одного элемента с указанным *IdЭлемента*, то он может выбрать любой из них или вернуть `null`.

В документах HTML этот метод ищет атрибут с именем `id`. Однако в XML-документах он ищет любой атрибут, имеющий тип `id`, независимо от того, какое имя имеет этот атрибут. Если типы XML-атрибутов неизвестны (например, если XML-анализатор не смог найти DTD документа), этот метод всегда возвращает `null`.

Этот метод важен и часто используется, так как обеспечивает простой способ получения объекта Element, представляющего определенный элемент в документе. Обратите внимание, что он предоставляет функциональность, аналогичную нестандартному массиву `document.all[]`, определенному в Internet Explorer 4 и более поздних версиях. И наконец, заметьте, что имя этого метода заканчивается на «Id», а не на «ID», и будьте аккуратны в его написании.

См. также

Document.getElementsByTagName(), Element.getElementsByTagName(), HTMLDocument.getElementsByName()

Document.getElementsByTagName() DOM Level 1 Core

получает все узлы Element с указанным именем тега

Синтаксис

```
Node[] getElementsByTagName(String имяТега);
```

Аргументы

имяТега

Имя тега для узлов Element, которые нужно получить, или строка маски «*» для получения всех узлов Element в документе независимо от имени тега. Для HTML-документов имена тегов сравниваются без учета регистра.

Возвращаемое значение

Доступный только для чтения массив (формально NodeList) всех узлов Element в дереве документа с указанным именем тега. Узлы Element возвращаются в том же порядке, в котором они расположены в исходном тексте документа.

Описание

Этот метод возвращает NodeList (его можно рассматривать как массив, доступный только для чтения), содержащий все узлы Element из документа, имеющие указанное имя тега, в том порядке, что и в исходном тексте документа. NodeList – «живой», то есть его содержимое по необходимости автоматически обновляется, если элементы с указанным именем тега добавляются или удаляются из документа.

Документы HTML нечувствительны к регистру, поэтому можно задавать *имяТега* в любой капитализации. С другой стороны, XML-документы чувствительны к регистру, и *имяТега* будет соответствовать только тегам, имеющим в исходном тексте документа то же имя и в точности ту же капитализацию.

Обратите внимание, что интерфейсы Element определяют метод с тем же именем, который выполняет поиск по поддереву документа. Кроме того, HTMLDocument определяет метод `getElementsByName()`, выполняющий поиск элементов по значению их атрибутов `name`, а не по именам тегов.

Пример

Поиск и перебор в цикле всех тегов `<h1>` в документе осуществляется так:

```
var headings = document.getElementsByTagName("h1");
for(var i = 0; i < headings.length; i++) {  // Цикл по всем полученным тегам
    var h = headings[i];
    // Теперь делаем что-нибудь с элементом <h1> в переменной h
}
```

См. также

Document.getElementById(), Element.getElementsByTagName(), HTMLDocument. getElementsByName()

Document.getElementsByTagNameNS() DOM Level 2 Core

возвращает все узлы Element с указанным именем и пространством имен

Синтаксис

```
Node[] getElementsByTagNameNS(String URIпространствоИмен,
                              String локальноеИмя);
```

Аргументы

URIпространствоИмен

Уникальный идентификатор пространства имен для требуемых элементов или «*» для соответствия любому пространству имен.

локальноеИмя

Локальное имя требуемых элементов или «*» для соответствия любому локальному имени.

Возвращаемое значение

Доступный только для чтения массив (формально NodeList) всех узлов Element в дереве документа, имеющих указанное пространство имен и локальное имя.

Описание

Этот метод работает точно так же, как `getElementsByTagName()`, за исключением того, что ищет элементы по пространству имен и имени. Его применение имеет смысл только в XML-документах, использующих пространства имен.

Document.getOverrideStyle() DOM Level 2 CSS

получает перекрывающий стиль для указанного элемента

Синтаксис

```
CSSStyleDeclaration getOverrideStyle(Element elt,
                                     String pseudoElt);
```

Аргументы

elt

Элемент, для которого запрашивается перекрывающий стиль.

pseudoElt

Псевдоэлемент для *elt* или `null`, если он отсутствует.

Возвращаемое значение

Объект CSSStyleDeclaration, представляющий информацию о перекрывающем стиле для указанного элемента и псевдоэлемента. Возвращаемый объект обычно также реализует более часто используемые интерфейсы CSS2Properties.

Описание

Этот метод возвращает объект CSSStyleDeclaration (который обычно также реализует CSS2Properties) для указанного элемента и (необязательно) псевдоэлемента. Полученным объектом можно воспользоваться для внесения изменений в стиль отображения указанного элемента, не затрагивая встроенного стиля этого элемента и не изменяя таблиц стилей документа. По существу, возвращаемое значение представляет объявление стиля в «перекрывающей» таблице стилей, имеющей приоритет над все-

ми другими таблицами стилей и встроенными стилями (за исключением объявлений `!important` в пользовательской таблице стилей).

Обратите внимание, что этот метод определяется не интерфейсом Document, а интерфейсом DocumentCSS. Если реализация поддерживает модуль CSS, объект Document всегда реализует DocumentCSS и определяет этот метод.

См. также

CSSStyleDeclaration, CSS2Properties, AbstractView.getComputedStyle(), HTMLElement.style

Document.importNode() DOM Level 2 Core

копирует узел из одного документа для использования в данном документе

Синтаксис

```
Node importNode(Node импортируемыйУзел,
                boolean глубина)
    throws DOMException;
```

Аргументы

импортируемыйУзел

Узел, который должен импортироваться.

глубина

Если `true`, также рекурсивно копирует всех потомков *импортируемогоУзла*.

Возвращаемое значение

Копия *импортируемогоУзла* (и, возможно, всех его потомков), в котором свойство `ownerDocument` установлено равным данному документу.

Исключения

Этот метод генерирует исключение DOMException с кодом `NOT_SUPPORTED_ERR`, если *импортируемыйУзел* является узлом Document или DocumentType, так как эти типы узлов не могут быть импортированы.

Описание

Этот метод получает узел, определенный в другом документе, и возвращает копию узла, подходящую для вставки в данный документ. Если аргумент *глубина* равен `true`, также копируются все потомки узла. Исходный узел и его потомки никак не модифицируются. В полученной копии свойство `ownerDocument` установлено равным данному документу, а `parentNode` равно `null`, поскольку копия пока не вставлена в документ. Функции-подписчики событий, зарегистрированные в исходном узле или дереве, не копируются.

Когда импортируется узел Element, с ним импортируются только атрибуты, явно заданные в исходном документе. Когда импортируется узел Attr, его свойство `specified` автоматически устанавливается равным `true`.

См. также

Node.cloneNode()

DocumentCSS

См. Document

DocumentEvent

См. Document

DocumentFragment

DOM Level 1 Core

смежные узлы и их поддеревья **Node → DocumentFragment**

Описание

Интерфейс DocumentFragment представляет часть (фрагмент) документа. Если говорить конкретно, то он представляет один или несколько смежных узлов документа и всех их потомков. Узлы DocumentFragment никогда не являются частью дерева документа, а унаследованное свойство `parentNode` в них всегда равно `null`. Однако особенность узлов DocumentFragment делает их очень полезными: когда поступает запрос на вставку DocumentFragment в дерево документа, вставляется не сам узел DocumentFragment, а все его дочерние узлы. Благодаря этому DocumentFragment хорош в качестве временного хранилища для узлов, которые требуется вставить в документ все сразу. DocumentFragment также очень полезен для реализации операций вырезания, копирования и вставки, особенно в сочетании с интерфейсом Range.

Можно создать новый, пустой DocumentFragment посредством метода `Document.createDocumentFragment()` или использовать метод `Range.extractContents()` или `Range.cloneContents()` для получения DocumentFragment, содержащего фрагмент существующего документа.

См. также

Range

Возвращается из: Document.createDocumentFragment(), Range.cloneContents(), Range. extractContents()

DocumentRange

См. Document

DocumentStyle

См. Document

DocumentTraversal

См. Document

DocumentType

DOM Level 1 XML

DTD XML-документа **Node → DocumentType**

Свойства

`readonly NamedNodeMap entities`

Этот объект NamedNodeMap представляет собой список объектов Entity, объявленных в DTD, и позволяет запрашивать объекты Entity по имени. Обратите внимание, что он не включает параметрические сущности XML. Этот NamedNodeMap является неизменяемым – его содержимое не может быть модифицировано.

`readonly String internalSubset` *[DOM Level 2]*

Внутреннее подмножество DTD (то есть часть DTD, присутствующая в самом документе, а не во внешнем файле). Ограничивающие квадратные скобки внутреннего подмножества не являются частью возвращаемого значения. Если внутреннее подмножество не существует, это свойство равно `null`.

`readonly String name`

Имя типа документа. Это идентификатор, расположенный непосредственно за `<!DOCTYPE>` в начале XML-документа, и он совпадает с именем тега корневого элемента документа.

`readonly NamedNodeMap notaions`

Объект NamedNodeMap, который содержит объекты Notation, представляющие все нотации, объявленные в DTD. Он также допускает поиск объектов Notation по имени нотации. Этот NamedNodeMap неизменяем – его содержимое не может быть модифицировано.

`readonly String publicId` *[DOM Level 2]*

Открытый идентификатор внешнего подмножества DTD или `null`, если он не указан.

`readonly String systemId` *[DOM Level 2]*

Системный идентификатор внешнего подмножества DTD или `null`, если он не указан.

Описание

Этот редко используемый интерфейс представляет DTD XML-документа. Программистам, работающим исключительно с документами HTML, этот интерфейс никогда не потребуется.

Так как DTD не является частью содержимого документа, узлы DocumentType никогда не присутствуют в дереве документа. Если в XML-документе имеется DTD, узел DocumentType для этого DTD доступен через свойство `doctype` узла Document.

Узлы DocumentType неизменяемы и не могут никак модифицироваться.

См. также

Document, Entity, Notation

Тип: Document.doctype

Передается в: DOMImplementation.createDocument()

Возвращается из: DOMImplementation.createDocumentType()

DocumentView

См. Document

DOMException

DOM Level 1 Core

сигнализирует об исключениях или ошибках в базовых объектах DOM

Константы

Следующие константы определяют допустимые значения для свойства `code` объекта DOMException. Обратите внимание, что эти константы являются статическими свойствами DOMException, а не свойствами индивидуальных объектов исключений.

`unsigned short INDEX_SIZE_ERR = 1`

Ошибка границ диапазона массива или индекса в строке.

`unsigned short DOMSTRING_SIZE_ERR = 2`

Возвращаемый текст слишком велик для строки в данной реализации JavaScript.

`unsigned short HIERARCHY_REQUEST_ERR = 3`

Была предпринята попытка поместить узел в недопустимое место в иерархическом дереве документа.

`unsigned short WRONG_DOCUMENT_ERR = 4`

Попытка использовать узел в документе, отличном от документа, создавшего узел.

`unsigned short INVALID_CHARACTER_ERR = 5`

Используется недопустимый символ (например, в имени элемента).

`unsigned short NO_DATA_ALLOWED_ERR = 6`

Сейчас не используется.

`unsigned short NO_MODIFICATION_ALLOWED_ERR = 7`

Была предпринята попытка модифицировать узел, доступный только для чтения и не допускающий модификации. Узлы Entity, EntityReference, Notation и все их потомки доступны только для чтения.

`unsigned short NOT_FOUND_ERR = 8`

Узел не был найден там, где ожидалось.

`unsigned short NOT_SUPPORTED_ERR = 9`

Метод или свойство не поддерживается в текущей реализации DOM.

`unsigned short INUSE_ATTRIBUTE_ERR = 10`

Была предпринята попытка связать узел Attr с узлом Element, когда узел Attr уже связан с другим Element.

`unsigned short INVALID_STATE_ERR = 11` *[DOM Level 2]*

Попытка использовать объект, который еще (или уже) не находится в состоянии, допускающем подобное использование.

`unsigned short SYNTAX_ERR = 12` *[DOM Level 2]*

Указанная строка содержит синтаксическую ошибку. Обычно используется со спецификациями свойств CSS.

`unsigned short INVALID_MODIFICATION_ERR = 13` *[DOM Level 2]*

Попытка модифицировать тип объекта CSSRule или CSSValue.

`unsigned short NAMESPACE_ERR = 14` *[DOM Level 2]*

Ошибка, связанная с пространствами имен элемента или атрибута.

`unsigned short INVALID_ACCESS_ERR = 15` *[DOM Level 2]*

Попытка доступа к объекту способом, который не поддерживается в данной реализации.

Свойства

`unsigned short code`

Код ошибки, предоставляющий некоторые сведения о причине исключения. Допустимые значения (и их смысл) для этого свойства определяются только что перечисленными константами.

Описание

Объект DOMException создается, когда метод или свойство DOM применяется некорректно или в неверном контексте. Значение свойства `code` обозначает общий тип возникшего исключения. Обратите внимание, что исключение DOMException может генерироваться при чтении или установке свойства объекта, а также при вызове метода объекта.

Описание свойств и методов объекта в этом справочнике включает список типов исключений, которые они могут генерировать. Обратите, однако, внимание, что некоторые часто возникающие исключения отсутствуют в этих списках. DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR` генерируется при каждой попытке модифицировать доступный только для чтения узел, такой как узел Entity или любой из его потомков. Следовательно, большинство методов и доступных для чтения и записи свойств интерфейса Node (и его подынтерфейсов) могут генерировать это исключение. Так как узлы, доступные только для чтения, присутствуют только в XML-документах, а не в документах HTML, и так как этот тип DOMException столь универсален для методов и доступных только для чтения свойств объектов Node, исключение `NO_MODIFICATION_ALLOWED_ERR` не вошло в описания этих методов и свойств.

Аналогично, многие методы и свойства DOM, возвращающие строки, могут генерировать DOMException с кодом `DOMSTRING_SIZE_ERR`, обозначающим, что возвращаемый текст слишком велик для представления в виде строкового значения в текущей реализации JavaScript. Несмотря на то что этот тип исключения теоретически может генерироваться многими свойствами и методами, на практике он встречается очень редко и опускается в описаниях этих методов и свойств.

Заметьте, что DOMException сигнализирует не обо всех исключениях в DOM. Исключения, относящиеся к событиям и обработчиками событий, приводят к созданию объекта EventException, а исключения, затрагивающие модуль DOM Range, приводят к возникновению исключения RangeException.

См. также

EventException, RangeException

DOMImplementation DOM Level 1 Core

методы, не зависящие от конкретного документа

Также реализует

`DOMImplementationCSS, HTMLDOMImplementation`

Если реализация DOM поддерживает модули HTML и CSS, то объект DOMImplementation также реализует интерфейсы DOMImplementationCSS и HTMLDOMImplementation. Из соображений удобства методы этих тривиальных интерфейсов перечислены здесь вместе с основными методами DOMImplementation.

Методы

`createCSSStyleSheet()` *[DOM Level 2 CSS]*

Метод DOMImplementationCSS создает новый объект CSSStyleSheet.

`createDocument()` *[DOM Level 2]*

Создает новый объект Document с корневым элементом (свойство `documentElement` возвращаемого объекта Document) указанного типа.

`createDocumentType()` *[DOM Level 2]*

Создает новый узел DocumentType.

`createHTMLDocument()` *[DOM Level 2 HTML]*

Метод HTMLDOMImplementation создает новый объект HTMLDocument и заполняет его элементами `<html>`, `<head>`, `<title>` и `<body>`.

`hasFeature()`

Проверяет, поддерживает ли текущая реализация указанную версию указанного модуля.

Описание

Интерфейс DOMImplementation и его подынтерфейсы HTMLDOMImplementation и DOMImplementationCSS представляют собой структуры, предназначенные для методов, не относящихся к какому-либо конкретному объекту Document, а являющихся «глобальными» для реализации DOM. Ссылку на объект DOMImplementation можно получить через свойство `implementation` любого объекта Document.

См. также

Тип: Document.implementation

DOMImplementation.createCSSStyleSheet() DOM Level 2 CSS

создает объект CSSStyleSheet

Синтаксис

```
CSSStyleSheet createCSSStyleSheet(String заголовок,
                                  String носитель)
    throws DOMException;
```

Аргументы

заголовок

Заголовок таблицы стилей.

носитель

Список типов носителей через запятую, к которым должна применяться таблица стилей.

Возвращаемое значение

Объект CSSStyleSheet.

Исключения

DOMException с кодом `SYNTAX_ERR`, если аргумент *носитель* имеет некорректный формат.

Описание

Этот метод создает новый объект CSSStyleSheet. Заметьте, однако, что в стандарте DOM до Level 2 пока еще не определен способ связывания только что созданного объекта CSSStyleSheet с документом.

Метод `createCSSStyleSheet()` определяется не интерфейсом DOMImplementation, а его подынтерфейсом DOMImplementationCSS. Если реализация поддерживает модуль «CSS», то его объект DOMImplementation реализует этот метод.

DOMImplementation.createDocument() DOM Level 2 Core

создает новый документ и указанный корневой элемент

Синтаксис

```
Document createDocument(String URIпространствоИмен,
                        String уточненноеИмя,
                        DocumentType типДокумента)
    throws DOMException;
```

Аргументы

URIпространствоИмен

Уникальный идентификатор пространства имен корневого элемента, который должен быть создан, или `null` в случае отсутствия пространства имен.

уточненноеИмя

Имя корневого элемента, который должен быть создан для этого документа. Если *URIпространствоИмен* не равно `null`, это имя должно включать префикс – имя пространства имен и двоеточие.

типДокумента

Объект DocumentType для только что созданного документа или `null`, если он не требуется.

Возвращаемое значение

Объект Document со свойством `documentElement`, установленным равным корневому узлу Element указанного типа.

Исключения

Этот метод может генерировать DOMException с перечисленными ниже значениями `code` в следующих ситуациях:

`INVALID_CHARACTER_ERR`

уточненноеИмя содержит недопустимый символ.

NAMESPACE_ERR

Формат *уточненногоИмени* неверен или есть несоответствие между *уточненнымИменем* и *URIпространстваИмен*.

NOT_SUPPORTED_ERR

Текущая реализация не поддерживает XML-документы и не реализует этот метод.

WRONG_DOCUMENT_ERR

типДокумента уже используется для другого документа или был создан с помощью другого объекта DOMImplementation.

Описание

Этот метод создает новый объект Document и указанный корневой объект `documentElement` для этого документа. Если аргумент *типДокумента* не равен `null`, свойство `ownerDocument` этого объекта DocumentType устанавливается равным только что созданному документу.

Этот метод применяется для создания XML-документа и может не поддерживаться реализациями, рассчитанными только на HTML. Для создания нового HTML-документа предназначен метод `createHTMLDocument()`.

См. также

createDocumentType(), createHTMLDocument()

DOMImplementation.createDocumentType() DOM Level 2 Core

создает новый узел DocumentType

Синтаксис

```
DocumentType createDocumentType(String уточненноеИмя,
                                String publicId,
                                String systemId)
    throws DOMException;
```

Аргументы

уточненноеИмя

Имя типа документа. Если используются пространства имен XML, это может быть уточненное имя, содержащее префикс пространства имен и локальное имя, разделенные двоеточием.

publicId

Открытый идентификатор типа документа или `null`.

systemId

Системный идентификатор типа документа или `null`. Этот аргумент обычно задает локальное имя файла DTD.

Возвращаемое значение

Новый объект `DocumentType` со свойством `ownerDocument`, равным `null`.

Исключения

Этот метод может генерировать DOMException с перечисленными ниже значениями `code` в следующих ситуациях:

INVALID_CHARACTER_ERR

уточненноеИмя содержит недопустимый символ.

NAMESPACE_ERR

Формат *уточненногоИмени* неверен.

NOT_SUPPORTED_ERR

Текущая реализация не поддерживает XML-документы и не реализует этот метод.

Описание

Этот метод создает новый узел DocumentType и задает только внешнее подмножество типа документа. Стандарт DOM до Level 2 не предоставляет какого-либо способа задания внутреннего подмножества, а в возвращаемом DocumentType не определены никакие узлы Entity или Notation. Этот метод полезен только для XML-документов и может не поддерживаться реализациями, ориентированными только на HTML.

DOMImplementation.createHTMLDocument() DOM Level 2 HTML

создает скелет HTML-документа

Синтаксис

```
HTMLDocument createHTMLDocument(String заголовок);
```

Аргументы

заголовок

Заголовок документа. Этот текст выступает в качестве содержимого элемента `<title>` только что созданного документа.

Возвращаемое значение

Новый объект HTMLDocument.

Описание

Этот метод создает новый объект HTMLDocument со скелетом дерева документа, включающего указанный заголовок. Свойство `documentElement` возвращаемого объекта – это элемент `<html>`, а в качестве дочерних элементов этот корневой элемент имеет теги `<head>` и `<body>`. В элементе `<head>`, в свою очередь, имеется дочерний элемент `<title>`, дочерним элементом которого является указанная строка *заголовок*.

Метод `createHTMLDocument()` определяется не интерфейсом DOMImplementation, а его подынтерфейсом HTMLDOMImplementation. Если реализация поддерживает модуль «HTML», то ее объект DOMImplementation реализует этот метод.

См. также

DOMImplementation.createDocument()

DOMImplementation.hasFeature() DOM Level 1 Core

определяет, поддерживает ли реализация определенный модуль

Синтаксис

```
boolean hasFeature(String модуль,
                   String версия);
```

Аргументы

модуль

Имя модуля (feature), поддержка которого проверяется. Набор допустимых имен модулей стандарта DOM Level 2 перечислен в приведенной ниже таблице. Имена модулей чувствительны к регистру.

версия

Номер версии модуля, поддержка которой проверяется, либо `null` или пустая строка `""`, если достаточно поддержки любой версии модуля. В спецификации DOM Level 2 поддерживаются номера версий 1.0 и 2.0.

Возвращаемое значение

`true`, если реализация полностью поддерживает указанную версию указанного модуля, или `false` в противном случае. Если номер версии не указан, метод возвращает `true`, если реализация полностью поддерживает любую версию указанного модуля.

Описание

Стандарт W3C DOM является модульным, и от реализаций не требуется реализовывать все модули или возможности стандарта. Этот метод проверяет, поддерживает ли реализация DOM указанный модуль. Информация о доступности в каждой статье этого справочника по DOM включает имя модуля. Обратите внимание, что хотя Internet Explorer 5 и 5.5 частично поддерживают спецификацию DOM Level 1, этот важный метод не поддерживался до IE 6.

Полный набор имен модулей, которые могут выступать в качестве аргумента *модуль*, показан в следующей таблице.

Модуль	Описание
Core	Реализованы Node, Element, Document, Text и другие фундаментальные интерфейсы, обязательные для всех реализаций DOM. Все совместимые реализации должны поддерживать этот модуль.
HTML	Реализованы HTMLElement, HTMLDocument и другие интерфейсы, специфические для HTML.
XML	Реализованы Entity, EntityReference, ProcessingInstruction, Notation и другие типы узлов, полезные только для XML-документов.
StyleSheets	Реализованы простые интерфейсы, описывающие обобщенные таблицы стилей.
CSS	Реализованы интерфейсы, специфические для таблиц стилей CSS.
CSS2	Реализован интерфейс CSS2Properties.
Events	Реализованы интерфейсы базовой обработки событий.
UIEvents	Реализованы интерфейсы для событий пользовательского интерфейса.
MouseEvents	Реализованы интерфейсы для событий мыши.
HTMLEvents	Реализованы интерфейсы для HTML-событий
MutationEvents	Реализованы интерфейсы для событий изменения документа.
Range	Реализованы интерфейсы для манипуляции с областями документа.
Traversal	Реализованы интерфейсы для расширенного обхода документа.
Views	Реализованы интерфейсы для представлений документа.

Пример

Этот метод может применяться в следующем коде:

```
// Проверяет, поддерживает ли броузер DOM Level 2 Traversal API
if (document.implementation &&
    document.implementation.hasFeature &&
    document.implementation.hasFeature("Traversal", "2.0")) {
  // Если да, используем его здесь...
}
else {
  // Если нет, выполняем обход документа каким-либо другим способом
}
```

См. также

Node.isSupported()

DOMImplementationCSS

См. DOMImplementation

Element

DOM Level 1 Core

Элемент HTML или XML **Node → Element**

Подынтерфейсы

HTMLElement

Свойства

`readonly String tagName`

Имя тега элемента. Например, для HTML-элемента `<p>` это строка «P». Для элементов HTML имя тега возвращается в верхнем регистре, независимо от его капитализации в исходном тексте документа. XML-документы чувствительны к регистру, и имя тега возвращается в точности в том виде, в каком оно написано в исходном тексте документа. Это свойство имеет то же значение, что и свойство `nodeName` интерфейса Node.

Методы

`getAttribute()`

Возвращает значение атрибута с указанным именем в виде строки.

`getAttributeNS()` *[DOM Level 2]*

Возвращает строковое значение атрибута, заданного локальным именем и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

`getAttributeNode()`

Возвращает значение атрибута с указанным именем в виде узла Attr.

`getAttributeNodeNS()` *[DOM Level 2]*

Возвращает значение Attr для атрибута, заданного локальным именем и URI пространства имен. Применение метода имеет смысл только для документов XML, использующих пространства имен.

`getElementsByTagName()`

Возвращает массив (формально NodeList) всех узлов Element, являющихся потомками данного элемента и имеющих указанное имя тега, в том порядке, в котором они расположены в документе.

`getElementsByTagNameNS()` *[DOM Level 2]*

То же самое, что и `getElementsByTagName()`, кроме того, что имя тега элемента задается локальным именем и URI пространства имен. Применение метода имеет смысл только для документов XML, использующих пространства имен.

`hasAttribute()` *[DOM Level 2]*

Возвращает `true`, если этот элемент имеет атрибут с указанным именем, или `false` в противном случае. Обратите внимание, что этот метод возвращает `true`, если указанный атрибут явно задан в исходном тексте документа или если DTD документа задает для этого атрибута значение по умолчанию.

`hasAttributeNS()` *[DOM Level 2]*

То же самое, что и `hasAttribute()`, за исключением того, что атрибут задается комбинацией локального имени и URI пространства имен. Применение метода имеет смысл только для документов XML, использующих пространства имен.

`removeAttribute()`

Удаляет из элемента атрибут с указанным именем. Заметьте, однако, что этот метод удаляет только атрибуты, явно заданные для этого элемента в исходном тексте документа. Если в DTD задается значение по умолчанию для этого атрибута, то это значение становится новым значением атрибута.

`removeAttributeNS()` *[DOM Level 2]*

То же самое, что и `removeAttribute()`, за исключением того, что атрибут, который должен быть удален, задается комбинацией локального имени и URI пространства имен. Применение метода имеет смысл только для документов XML, использующих пространства имен.

`removeAttributeNode()`

Удаляет указанный узел Attr из списка атрибутов этого элемента. Обратите внимание, что этот метод работает только в отношении атрибутов, явно указанных для этого элемента в исходном тексте документа. Если DTD задает значение по умолчанию для удаляемого атрибута, создается новый узел Attr для значения атрибута по умолчанию.

`setAttribute()`

Устанавливает атрибут с указанным именем равным указанному значению. Если атрибут с таким именем еще не существует, в элемент добавляется новый атрибут.

`setAttributeNS()` *[DOM Level 2]*

То же самое, что и `setAttribute()`, за исключением того, что атрибут, который должен быть установлен, задается комбинацией локального имени и URI пространства имен. Применение метода имеет смысл только для XML-документов, использующих пространства имен.

`setAttributeNode()`

Добавляет указанный узел Attr в список атрибутов этого элемента. Если атрибут с тем же именем уже существует, его значение заменяется.

`setAttributeNodeNS()` *[DOM Level 2]*

То же самое, что и `setAttributeNode()`, но этот метод подходит для узлов, возвращаемых методом `Document.createAttributeNS()`. Применение метода имеет смысл только для документов XML, использующих пространства имен.

Описание

Интерфейс Element представляет элементы (теги) HTML или XML. Свойство `tagName` задает имя элемента. Метод `getElementsByTagName()` предоставляет мощное средство для нахождения элементов-потомков данного элемента, имеющих указанное имя тега. Другие методы этого интерфейса предоставляют доступ к атрибутам элемента. Если сопоставить элементу уникальный идентификатор, используя атрибут `id` в исходном тексте документа, то нетрудно будет найти узел Element, представляющий данный элемент документа с помощью полезного метода `Document.getElementById()`. Для создания нового узла Element, предназначенного для вставки в документ, применяется метод `Document.createElement()`.

В документах HTML (и во многих XML-документах) все атрибуты имеют простые строковые значения, и для любых манипуляций с атрибутами можно применять простые методы `getAttribute()` и `setAttribute()`.

При работе с XML-документами, которые могут содержать ссылки на сущности как часть значений атрибутов, вам придется иметь дело с объектами Attr и их поддеревом узлов. Можно получать и устанавливать объект Attr для атрибута с помощью методов `getAttributeNode()` и `setAttributeNode()` или выполнять цикл по узлам Attr в массиве `attributes[]` интерфейса Node. Для работы с документом XML, использующим пространства имен XML, предназначены методы, имена которых заканчиваются на «NS».

В спецификации DOM Level 1 метод `normalize()` был частью интерфейса Element. В спецификации Level 2 `normalize()` стал частью интерфейса Node. Все узлы Element наследуют этот метод, и все равно могут его использовать.

См. также

HTMLElement, Node

Тип: Attr.ownerElement, Document.documentElement

Передается в: Document.getOverrideStyle(), AbstractView.getComputedStyle()

Возвращается из: Document.createElement(), Document.createElementNS(), Document. getElementById()

Element.getAttribute() DOM Level 1 Core

возвращает строковое значение атрибута с указанным именем

Синтаксис

```
String getAttribute(String имя);
```

Аргументы

имя

Имя атрибута, значение которого должно быть возвращено.

Возвращаемое значение

Строковое значение атрибута с указанным именем. Если в документе не указано значение атрибута, и типом документа не задается значение по умолчанию, возвращаемым значением будет пустая строка (`""`).

Описание

Метод getAttribute() возвращает значение указанного атрибута для элемента. В документах HTML значения атрибутов всегда являются строками, и этот метод всегда возвращает полное значение атрибута. Обратите внимание, что объекты, представляющие HTML-элементы, также реализуют интерфейс HTMLElement и один из его подынтерфейсов, специфических для конкретного тега. Следовательно, все стандартные атрибуты стандартных тегов HTML также непосредственно доступны как свойства объекта Element.

В XML-документах значения атрибутов недоступны непосредственно как свойства элемента, и к ним надо обращаться путем вызова метода. Для многих XML-документов подходящим для этого методом является getAttribute(). Обратите, однако, внимание, что в атрибутах XML могут содержаться ссылки на сущности, и чтобы получить всю информацию о таких атрибутах, необходимо вызвать getAttributeNode(), возвращающий узел Attr, поддерево которого представляет значение атрибута целиком. Узлы Attr для элемента также доступны в массиве attributes[], унаследованном от интерфейса Node. Для документов XML может потребоваться метод getAttributeNS() или getAttributeNodeNS().

Пример

Следующий код иллюстрирует два различных способа получения значения атрибута для элемента HTML <img>:

```
// Получаем все изображения в документе
var images = document.body.getElementsByTagName("IMG");
// Получаем атрибут SRC первого изображения
var src0 = images[0].getAttribute("SRC");
// Получаем атрибут SRC второго изображения путем чтения свойства
var src1 = images[1].src;
```

См. также

Element.getAttributeNode(), Node.attributes

Element.getAttributeNode() DOM Level 1 Core

возвращает узел Attr для атрибута с указанным именем

Синтаксис

Attr getAttributeNode(String *имя*);

Аргументы

имя

Имя требуемого атрибута.

Возвращаемое значение

Узел Attr, потомки которого представляют значение атрибута с указанным именем, или null, если в этом элементе нет такого атрибута.

Описание

Метод getAttributeNode() возвращает узел Attr, представляющий значение атрибута с указанным именем. Заметьте, что этот узел Attr может быть также получен через свойство attributes, унаследованное от интерфейса Node.

Значение атрибута представлено потомками узла Attr. В узле Attr документов HTML имеется только один дочерний узел Text, и всегда проще получить значение атрибута, вызвав метод `getAttribute()`, который возвращает значение в виде строки. Метод `getAttributeNode()` нужен только при работе с документами XML, содержащими в значениях своих атрибутов ссылки на сущности.

См. также

Element.getAttribute(), Element.getAttributeNodeNS(), Node.attributes

Element.getAttributeNodeNS() DOM Level 2 Core

возвращает узел Attr для атрибута с пространством имен

Синтаксис

```
Attr getAttributeNodeNS(String URIпространстваИмен,
                        String локальноеИмя);
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен этого атрибута, или `null`, если пространства имен нет.

локальноеИмя

Идентификатор, задающий имя атрибута в его пространстве имен.

Возвращаемое значение

Узел Attr, потомки которого представляют значение указанного атрибута, или `null`, если в этом элементе нет такого атрибута.

Описание

Этот метод работает так же, как `getAttributeNode()`, за исключением того, что атрибут задается комбинацией URI пространства имен и локального имени, определенного в данном пространстве имен. Применение этого метода имеет смысл только для документов XML, использующих пространства имен.

См. также

Element.getAttributeNode(), Element.getAttributeNS()

Element.getAttributeNS() DOM Level 2 Core

получает значение атрибута, использующего пространства имен

Синтаксис

```
String getAttributeNS(String URIпространстваИмен,
                      String локальноеИмя);
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен этого атрибута, или `null`, если пространства имен нет.

локальноеИмя

Идентификатор, задающий имя атрибута в его пространстве имен.

Возвращаемое значение

Строковое значение атрибута с указанным именем. Если в документе не указано значение атрибута и типом документа не задается значение по умолчанию, возвращаемым значением будет пустая строка.

Описание

Этот метод работает точно так же, как метод `getAttribute()`, кроме того, что атрибут задается комбинацией URI пространства имен и локального имени, определенного в данном пространстве имен. Применение этого метода имеет смысл только для документов XML, использующих пространства имен.

См. также

Element.getAttribute(), Element.getAttributeNodeNS()

Element.getElementsByTagName() DOM Level 1 Core

находит элементы-потомки с указанным именем тега

Синтаксис

```
Node[] getElementsByTagName(String имя);
```

Аргументы

имя

Имя тега требуемых элементов или значение «*», обозначающее, что должны возвращаться все элементы-потомки независимо от имени тега.

Возвращаемое значение

Массив (формально NodeList) объектов Element, являющихся потомками этого элемента и имеющих указанное имя тега.

Описание

Этот метод обходит всех потомков этого элемента и возвращает массив (фактически объект NodeList) узлов Element, представляющих все элементы документа с указанным именем тега. Элементы в возвращаемом массиве расположены в том же порядке, в каком они присутствуют в исходном документе.

Обратите внимание, что в интерфейсе Document также имеется метод `getElementsByTagName()`, который работает так же, как данный, но обходит весь документ, а не только потомков одного элемента. Не путайте этот метод с `HTMLDocument.getElementsByName()`, который выполняет поиск элементов на основе значения их атрибута `name`, а не имени тега.

Пример

Все теги `<div>` в документе позволяет найти следующий код:

```
var divisions = document.body.getElementsByTagName("div");
```

А этот код может найти все теги `<p>` внутри первого тега `<div>`:

```
var paragraphs = divisions[0].getElementsByTagname("p");
```

См. также

Document.getElementById(), Document.getElementsByTagName(), HTMLDocument. getElementsByName()

Element.getElementsByTagNameNS() DOM Level 2 Core

возвращает элементы-потомки с указанным именем и пространством имен

Синтаксис

```
Node[] getElementsByTagNameNS(String URIпространствоИмен,
                              String локальноеИмя);
```

Аргументы

URIпространствоИмен

URI, уникально идентифицирующее пространство имен элемента.

локальноеИмя

Идентификатор, задающий имя элемента внутри пространства имен.

Возвращаемое значение

Массив (формально NodeList) объектов Element, являющихся потомками этого элемента и имеющих указанное имя и пространство имен.

Описание

Этот метод работает так же, как `getElementsByTagName()`, но имя тега требуемого элемента задается в виде комбинации URI пространства имен и локального имени, определенного внутри пространства имен. Применение этого метода имеет смысл только при работе с документами XML, использующими пространства имен.

См. также

Document.getElementsByTagNameNS(), Element.getElementsByTagName()

Element.hasAttribute() DOM Level 2 Core

определяет, имеется ли в этом элементе указанный атрибут

Синтаксис

```
boolean hasAttribute(String имя);
```

Аргументы

имя

Имя требуемого атрибута.

Возвращаемое значение

`true`, если этот элемент имеет заданное значение или значение по умолчанию для указанного атрибута, и `false` в противном случае.

Описание

Этот метод определяет, имеется ли в элементе атрибут с указанным именем, но не возвращает значение этого атрибута. Обратите внимание, что `hasAttribute()` возвращает `true`, если указанный атрибут явно задан в документе, а также если тип документа задает значение по умолчанию для этого атрибута.

См. также

Attr.specified, Element.getAttribute(), Element.setAttribute()

Element.hasAttributeNS() DOM Level 2 Core

определяет, имеется ли в этом элементе указанный атрибут

Синтаксис

```
boolean hasAttributeNS(String URIпространствоИмен,
                       String локальноеИмя);
```

Аргументы

URIпространствоИмен

Уникальный идентификатор пространства имен для атрибута или `null` в случае отсутствия пространства имен.

локальноеИмя

Имя атрибута внутри указанного пространства имен.

Возвращаемое значение

`true`, если в этом элементе есть явно заданное значение или значение по умолчанию для указанного атрибута; в противном случае `false`.

Описание

Этот метод работает точно так же, как `hasAttribute()`, за исключением того, что проверяемый атрибут задается пространством имен и именем. Применение этого метода имеет смысл только в документах XML, использующих пространства имен.

См. также

Element.getAttributeNS(), Element.setAttributeNS()

Element.removeAttribute() DOM Level 1 Core

удаляет атрибут с указанным именем из элемента

Синтаксис

```
void removeAttribute(String имя);
```

Аргументы

имя

Имя атрибута, который должен быть удален.

Исключения

Этот метод может генерировать DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если элемент доступен только для чтения и не допускает удаления своих атрибутов.

Описание

`removeAttribute()` удаляет атрибут с указанным именем из этого элемента. Если для указанного атрибута типом документа задается значение по умолчанию, последующие вызовы `getAttribute()` будут возвращать значение по умолчанию. Попытки удалить несуществующие атрибуты или атрибуты, которые не указаны, но имеют значения по умолчанию, молча игнорируются.

См. также

Element.getAttribute(), Element.setAttribute(), Node.attributes

Element.removeAttributeNode() DOM Level 1 Core

удаляет узел Attr из элемента

Синтаксис

```
Attr removeAttributeNode(Attr старыйАтрибут)
    throws DOMException;
```

Аргументы

старыйАтрибут

Узел Attr, который должен быть удален из элемента.

Возвращаемое значение

Удаленный узел Attr.

Исключения

Этот метод может генерировать DOMException со следующими значениями `code`:

`NO_MODIFICATION_ALLOWED_ERR`

Этот элемент доступен только для чтения и не допускает удаления атрибутов.

`NOT_FOUND_ERR`

старыйАтрибут не является атрибутом этого элемента.

Описание

Этот метод удаляет (и возвращает) узел Attr из набора атрибутов элемента. Если для удаляемого атрибута в DTD задано значение по умолчанию, добавляется новый узел Attr, представляющий значение по умолчанию. Вместо этого метода часто проще применить метод `removeAttribute()`.

См. также

Attr, Element.removeAttribute()

Element.removeAttributeNS() DOM Level 2 Core

удаляет атрибут, заданный именем и пространством имен

Синтаксис

```
void removeAttributeNS(String URIпространстваИмен,
                       String локальноеИмя);
```

Аргументы

URIпространстваИмен

Уникальный идентификатор для пространства имен атрибута или `null` в случае отсутствия пространства имен.

локальноеИмя

Имя атрибута в указанном пространстве имен.

Исключения

Этот метод может генерировать DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если этот элемент доступен только для чтения и не допускает удаления своих атрибутов.

Описание

Метод `removeAttributeNS()` работает точно так же, как `removeAttribute()`, за исключением того, что удаляемый атрибут задается именем и пространством имен, а не просто именем. Применение этого метода имеет смысл только для XML-документов, использующих пространство имен.

См. также

Element.getAttributeNS(), Element.removeAttribute(), Element.setAttributeNS()

Element.setAttribute() DOM Level 1 Core

создает или изменяет значение атрибута элемента

Синтаксис

```
void setAttribute(String имя,
                  String значение)
    throws DOMException;
```

Аргументы

имя

Имя атрибута, который должен быть создан или модифицирован.

значение

Строковое значение атрибута.

Исключения

Этот метод может генерировать DOMException со следующими значениями `code`:

`INVALID_CHARACTER_ERR`

В аргументе *имя* есть символ, не допустимый в именах атрибутов HTML или XML.

`NO_MODIFICATION_ALLOWED_ERR`

Этот элемент доступен только для чтения и не допускает модификации своих атрибутов.

Описание

Этот метод устанавливает указанный атрибут равным указанному значению. Если атрибута с таким именем еще нет, создается новый атрибут. Обратите внимание, что объекты Element, представляющие теги HTML-документа, также реализуют интерфейс HTMLElement и (обычно) один из интерфейсов, специфических для конкретного тега. Для удобства эти интерфейсы определяют свойства, соответствующие стандартным атрибутам HTML каждого тега, и обычно проще установить атрибут HTML, просто присвоив значение соответствующему свойству.

Аргумент *значение* – это обычная строка. Если при работе с документом XML требуется включить в значение атрибута ссылку на сущность, используйте `setAttributeNode()`.

Пример

```
// Устанавливает атрибут TARGET всех ссылок в документе
var links = document.body.getElementsByTagName("A");
for(var i = 0; i < links.length; i++) {
    links[i].setAttribute("TARGET", "newwindow");
}
```

См. также

Element.getAttribute(), Element.removeAttribute(), Element.setAttributeNode()

Element.setAttributeNode() DOM Level 1 Core

добавляет к элементу новый узел Attr

Синтаксис

```
Attr setAttributeNode(Attr новыйАтрибут)
    throws DOMException;
```

Аргументы

новыйАтрибут

Узел Attr, представляющий атрибут, который должен быть добавлен или значение которого должно быть изменено.

Возвращаемое значение

Узел Attr, который был заменен *новымАтрибутом*, или `null`, если никакой атрибут не был заменен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code`:

`INUSE_ATTRIBUTE_ERR`

новыйАтрибут уже входит в набор атрибутов какого-либо другого узла Element.

`NO_MODIFICATION_ALLOWED_ERR`

Узел Element доступен только для чтения, и его атрибуты нельзя модифицировать.

`WRONG_DOCUMENT_ERR`

Значение свойства `ownerDocument` *новогоАтрибута* отлично от элемента, для которого оно устанавливается.

Описание

Этот метод добавляет новый узел Attr к набору атрибутов узла Element. Если атрибут с тем же именем для элемента уже существует, то *новыйАтрибут* заменяет этот атрибут, и возвращается замененный узел Attr. Если такого атрибута еще нет, этот метод определяет новый атрибут для элемента.

Обычно проще использовать метод `setAttribute()` вместо `setAttributeNode()`. Однако для определения атрибута, значение которого содержит ссылку на сущность для XML-документа, следует вызвать `setAttributeNode()`.

См. также

Attr, Element.setAttribute()

Element.setAttributeNodeNS() DOM Level 2 Core

добавляет узел Attr с пространством имен в узел Element

Синтаксис

```
Attr setAttributeNodeNS(Attr новыйАтрибут)
    throws DOMException;
```

Аргументы

новыйАтрибут

Узел Attr, представляющий атрибут, который должен быть добавлен или значение которого должно быть изменено.

Возвращаемое значение

Узел Attr, который был заменен *новымАтрибутом*, или `null`, если никакой атрибут не был заменен.

Исключения

Этот метод генерирует исключения по тем же причинам, что и `setAttributeNode()`. Он может также генерировать DOMException с кодом `NOT_SUPPORTED_ERR`, сигнализируя о том, что метод не реализован, потому что текущая реализация не поддерживает документы XML и пространства имен.

Описание

Этот метод работает точно так же, как `setAttributeNode()`, за исключением того, что он предназначен для использования с узлами Attr, представляющими атрибуты, задаваемые пространством имен и именем.

Применение этого метода имеет смысл только при работе с документами XML, использующими пространства имен. Он может не быть реализован (то есть генерировать исключение `NOT_SUPPORTED_ERR`) в броузерах, не поддерживающих XML-документы.

См. также

Attr, Element.setAttributeNS(), Element.setAttributeNode()

Element.setAttributeNS() DOM Level 2 Core

создает или изменяет атрибут с пространством имен

Синтаксис

```
void setAttributeNS(String URIпространстваИмен,
                    String уточненноеИмя,
                    String значение)
    throws DOMException;
```

Аргументы

URIпространстваИмен

URI, уникально идентифицирующий пространство имен атрибута, который должен быть установлен или создан, или `null` в случае отсутствия пространства имен.

уточненноеИмя

Имя атрибута, заданное в виде префикса пространства имен, за которым следуют двоеточие и имя внутри этого пространства имен.

значение

Новое значение атрибута.

Исключения

Этот метод может генерировать DOMException со следующими значениями `code`:

`INVALID_CHARACTER_ERR`

Аргумент *уточненноеИмя* содержит символ, не допустимый в именах атрибутов HTML или XML.

`NAMESPACE_ERR`

Формат *уточненногоИмени* неверен, или имеется несоответствие между префиксом пространства имен в *уточненномИмени* и аргументом *URIпространстваИмен*.

`NO_MODIFICATION_ALLOWED_ERR`

Этот элемент доступен только для чтения и не допускает модификации своих атрибутов.

`NOT_SUPPORTED_ERR`

Реализация DOM не поддерживает документы XML.

Описание

Этот метод похож на `setAttribute()` за исключением того, что создаваемый или устанавливаемый атрибут задается URI пространства имен и уточненным именем, состоящим из префикса пространства имен, двоеточия и локального имени внутри пространства имен. Кроме возможности изменения значения атрибута, этот метод позволяет изменить префикс пространства имен атрибута.

Применение этого метода имеет смысл только при работе с документами XML, использующими пространства имен. Он может не быть реализован (то есть генерировать исключение `NOT_SUPPORTED_ERR`) в броузерах, не поддерживающих XML-документы.

См. также

Element.setAttribute(), Element.setAttributeNode()

ElementCSSInlineStyle

См. HTMLElement

Entity

DOM Level 1 XML

сущность в DTD XML **Node → Entity**

Свойства

`readonly String notationName`

Имя нотации (для внешних неанализируемых сущностей) или `null`, если оно отсутствует (для анализируемых сущностей). Способ поиска узла Notation по имени рассмотрен в описании свойства `notations` интерфейса DocumentType.

`readonly String publicId`

Публичный идентификатор этой сущности или `null`, если он не указан.

`readonly String systemId`

Системный идентификатор этой сущности или `null`, если он не указан.

Описание

Этот редко используемый интерфейс представляет сущность в определении типа документа (DTD). Он никогда не применяется при работе с документами HTML.

Имя сущности задается свойством nodeName, унаследованным от интерфейса Node. Содержимое сущности представлено дочерними узлами узла Entity. Обратите внимание, что узлы Entity и их дочерние элементы не являются частью дерева документа (а свойство parentNode узла Entity всегда равно null). Вместо этого документ может содержать одну или несколько ссылок на сущности. Дополнительную информацию можно найти в статье «EntityReference».

Сущности определяются в DTD документа либо как часть внешнего файла DTD, либо как часть «внутреннего подмножества», определяющего локальные сущности, специфические для текущего документа. В интерфейсе DocumentType имеется свойство entities, позволяющее искать узлы Entity по имени. Это единственный способ получения ссылки на узел Entity; так как узлы Entity всегда являются частью определения типа документа, они никогда не бывают частью самого документа.

Узлы Entity и все их потомки доступны только для чтения и не могут каким-либо образом редактироваться или изменяться.

См. также

DocumentType, EntityReference, Notation

EntityReference

DOM Level 1 XML

ссылка на сущность, определенную в XML DTD **Node → EntityReference**

Описание

Этот редко используемый интерфейс представляет ссылку из документа XML на сущность, определенную в DTD документа. Символьные сущности и предопределенные сущности, такие как <, всегда разворачиваются в XML- и HTML-документах, и узлы EntityReference никогда не появляются в документах HTML, поэтому программистам, работающим исключительно с последними, этот интерфейс никогда не потребуется. Обратите внимание, что некоторые XML-анализаторы разворачивают все ссылки на сущности. Документы, созданные такими анализаторами, не содержат узлов EntityReference.

Этот интерфейс не определяет никаких собственных свойств или методов. Унаследованное свойство nodeName задает имя сущности, на которую указывает ссылка. Свойство entities интерфейса DocumentType предоставляет способ поиска объекта Entity с таким именем. Однако обратите внимание, что DocumentType может не содержать узла Entity с указанным именем (например, потому, что непроверяющие XML-анализаторы не обязаны анализировать «внешнее подмножество» DTD). В этом случае EntityReference не имеет дочерних узлов. С другой стороны, если DocumentType содержит узел Entity с указанным именем, то дочерние узлы EntityReference являются копиями дочерних узлов узла Entity и представляют содержимое сущности. Как и узлы Entity, узлы EntityReference и их потомки доступны только для чтения и не могут редактироваться и изменяться.

См. также

DocumentType

Возвращается из: Document.createEntityReference()

Event

DOM Level 2 Events

информация о событии

Подынтерфейсы

MutationEvent, UIEvent

Константы

Эти константы являются допустимыми значениями свойства `eventPhase`; они представляют текущую стадию распространения данного события:

`unsigned short CAPTURING_PHASE = 1`

Событие находится в стадии перехвата.

`unsigned short AT_TARGET = 2`

Событие обрабатывается своим целевым узлом.

`unsigned short BUBBLING_PHASE = 3`

Событие всплывает.

Свойства

`readonly boolean bubbles`

`true`, если тип события поддерживает всплывание (и если не вызван метод `stopPropagation()`), и `false` в противном случае.

`readonly boolean cancelable`

`true`, если действие по умолчанию, связанное с событием, может быть отменено с помощью метода `preventDefault()`; и `false` в противном случае.

`readonly EventTarget currentTarget`

Узел документа, обрабатывающий событие в данный момент. Во время перехвата и всплывания значение свойства отличается от `target`. Обратите внимание, что все узлы реализуют интерфейс EventTarget, и `currentTarget` может ссылаться на любой узел; значение этого свойства не ограничивается узлами Element.

`readonly unsigned short eventPhase`

Текущая стадия распространения события. Допустимые значения этого свойства определяются тремя приведенными выше константами.

`readonly EventTarget target`

Целевой узел этого события, то есть узел, сгенерировавший событие. Обратите внимание, что это может быть любой узел; значение этого свойства не ограничивается узлами Element.

`readonly Date timeStamp`

Дата и время, когда произошло событие (или, формально, когда был создан объект Event). Реализации не обязаны предоставлять допустимое значение времени в этом поле, и в этом случае метод `getTime()` данного объекта Date должен возвращать 0. См. описание объекта Date в «Справочнике по базовому JavaScript».

`readonly String type`

Имя события, представляемого данным объектом Event. Это имя, с которым был зарегистрирован обработчик события, или имя свойства обработчика события без ведущих символов «on». Например, «click», «load» или «submit». Полный список типов событий, определяемых стандартом DOM, см. в табл. 19.3.

Методы

`initEvent()`

Инициализирует свойства только что созданного объекта Event.

`preventDefault()`

Сообщает броузеру о том, чтобы он не выполнял действие по умолчанию, связанное с этим событием, если таковое существует. Если событие не относится к отменяемому типу, этот метод не выполняет никаких действий.

`stopPropagation()`

Прекращает дальнейшее распространение события через стадии перехвата, обработки и всплытия. После вызова этого метода любые другие обработчики того же события в том же узле будут вызваны, но событие не будет передано каким-либо другим узлам.

Описание

Этот интерфейс представляет событие, произошедшее в каком-либо узле документа, и содержит подробности о событии. Различные подынтерфейсы интерфейса Event определяют дополнительные свойства, предоставляющие сведения об определенных типах событий.

Многие типы событий используют более специфический подынтерфейс Event для описания произошедшего события. Однако типы событий, определенные модулем HTMLEvents, непосредственно используют интерфейс Event. Вот эти типы событий: «abort», «blur», «change», «error», «focus», «load», «reset», «resize», «scroll», «select», «submit» и «unload».

См. также

EventListener, EventTarget, MouseEvent, UIEvent; главу 19

Передается в: EventTarget.dispatchEvent()

Возвращается из: Document.createEvent()

Event.initEvent() DOM Level 2 Events

инициализирует свойство нового события

Синтаксис

```
void initEvent(String eventTypeArg,
               boolean canBubbleArg,
               boolean cancelableArg);
```

Аргументы

eventTypeArg

Тип события. Это может быть один из предопределенных типов событий, таких как «load» или «submit», или специальный тип, выбранный программистом. Однако имена, начинающиеся с «DOM», зарезервированы.

canBubbleArg

Будет ли событие всплывать.

cancelableArg

Может ли событие быть отменено с помощью метода `preventDefault()`.

Описание

Этот метод инициализирует свойства type, bubbles и cancelable синтетического объекта Event, созданного с помощью Document.createEvent(). Этот метод может вызываться для только что созданных объектов Event только до их диспетчеризации с помощью метода EventTarget.dispatchEvent().

Event.preventDefault() DOM Level 2 Events

отменяет действие по умолчанию для события

Синтаксис

```
void preventDefault();
```

Описание

Этот метод сообщает броузеру, чтобы он не выполнял связанное с этим событием действие по умолчанию (если оно есть). Например, если свойство type равно «submit», любой обработчик события, вызванный во время любой стадии обработки события, может отменить передачу данных формы, вызвав этот метод. Обратите внимание, что если свойство cancelable объекта Event равно false, то действие по умолчанию либо отсутствует, либо не может быть отменено. В любой из этих ситуаций вызов данного метода не будет иметь никакого эффекта.

Event.stopPropagation() DOM Level 2 Events

не передавать событие дальше

Синтаксис

```
void stopPropagation();
```

Описание

Этот метод останавливает распространение события и отменяет его передачу другим узлам документа. Он может вызываться во время любой стадии распространения события. Обратите внимание, что этот метод не отменяет вызов любых других обработчиков событий того же узла документа, но предотвращает передачу событий любым другим узлам.

EventException DOM Level 2 Events

сигнализирует о специфическом для события исключении или ошибке

Константы

Следующая константа определяет одно допустимое значение для свойства code объекта EventException. Обратите внимание, что эта константа является статическим свойством EventException, а не свойством индивидуальных объектов исключений.

```
unsigned short UNSPECIFIED_EVENT_TYPE_ERR = 0
```

У объекта Event есть свойство type, которое не было инициализировано либо равно null или пустой строке.

Свойства

`unsigned shomrt code`

Код ошибки, предоставляющий некоторые сведения о том, что привело к ошибке. В DOM Level 2 для этого поля есть только одно возможное значение, определенное константой, которая приведена выше.

Описание

EventException генерируется некоторыми методами, связанными с обработкой событий для уведомления о какой-либо проблеме. (В спецификации DOM Level 2 исключение этого типа генерируется только методом `EventTarget.dispatchEvent()`).

EventListener

DOM Level 2 Events

функция-обработчик события

Методы

`handleEvent()`

В таких языках, как Java, не допускающих передачу функций в качестве параметров другим функциям, программист определяет слушателей событий путем создания класса, реализующего этот интерфейс и включающего для этого реализацию метода `handleEvent()`. Когда происходит событие, система вызывает этот метод и передает ему объект Event, описывающий событие.

Однако в JavaScript вы определяете обработчик события, просто написав функцию, принимающую в качестве аргумента объект Event. Имя функции значения не имеет, и в роли объекта EventListener выступает сама функция. См. раздел «Пример».

Описание

Этот интерфейс определяет структуру слушателя или обработчика события. В языках, подобных Java, слушатель события – это объект, определяющий метод с именем `handleEvent()`, который принимает в качестве единственного аргумента объект Event. Однако в JavaScript любая функция, ожидающая единственный аргумент Event, или функция, не требующая аргументов, может выступать в качестве слушателя события.

Пример

```
// Эта функция является слушателем события "submit"
function submitHandler(e) {
  // Вызывает функцию проверки формы, определенной в другом месте
  if (!validate(e.target))
      e.preventDefault();  // Если проверка не проходит, форма не подтверждается
}

// Вот так мы можем зарегистрировать приведенный выше слушатель события
document.forms[0].addEventListener("submit", submitHandler, false);
```

См. также

Event, EventTarget;

Передается в: EventTarget.addEventListener(), EventTarget.removeEventListener()

EventTarget

DOM Level 2 Events

методы регистрации слушателя события

Методы

`addEventListener()`

Добавляет слушатель события в набор слушателей для данного узла.

`dispatchEvent()`

Передает этому узлу синтетическое событие.

`removeEventListener()`

Удаляет слушатель события из набора слушателей этого узла документа.

Описание

В реализациях DOM, поддерживающих события (то есть в тех, которые поддерживают модуль «Events»), все узлы в дереве документа реализуют этот интерфейс и поддерживают набор или список функций-слушателей событий для каждого узла. Методы `addEventListener()` и `removeEventListener()` позволяют добавлять и удалять функции-слушатели из этого набора.

См. также

Event, EventListener;

Тип: Event.currentTarget, Event.target, MouseEvent.relatedTarget

Передается в: MouseEvent.initMouseEvent()

EventTarget.addEventListener()

DOM Level 2 Events

регистрирует обработчик событий

Синтаксис

```
void addEventListener(String тип,
                      EventListener слушатель,
                      boolean использованиеПерехват);
```

Аргументы

тип

Тип события, для которого должен вызываться слушатель. Например, «load», «click» или «mousedown».

слушатель

Функция обработки события, которая должна вызываться при передаче события указанного типа в этот узел документа.

использованиеПерехват

Если этот аргумент равен `true`, указанный *слушатель* должен вызываться только во время стадии перехвата при распространении события. Чаще встречается значение `false`, обозначающее, что *слушатель* не будет вызываться на стадии перехвата, а будет вызваться, когда этот узел является фактической целью события или когда событие «всплывает» к этому узлу от своего первоначального целевого узла.

Описание

Этот метод добавляет указанную функцию-слушатель события к набору слушателей, зарегистрированных в этом узле для обработки событий указанного *типа*. Если аргумент *использоватьПерехват* равен `true`, слушатель регистрируется как перехватывающий слушатель события. Если *использоватьПерехват* равен `false`, он регистрируется как обычный обработчик события.

Метод `addEventListener()` может вызываться несколько раз для регистрации нескольких обработчиков событий для одного типа событий в одном узле. Заметьте, однако, что DOM никак не гарантирует порядок, в котором будут вызываться множественные обработчики событий.

Если одна и та же функция-слушатель события регистрируется дважды в том же узле с одинаковыми аргументами *тип* и *использоватьПерехват*, то вторая регистрация просто игнорируется. Если новый слушатель события регистрируется для этого узла во время обработки в нем события, новый слушатель не вызывается для данного события.

Если узел документа дублируется с помощью метода `Node.cloneNode()` или `Document.importNode()`, слушатели событий, зарегистрированные в исходном узле, не копируются.

См. также

Event, EventListener; главу 19

EventTarget.dispatchEvent() DOM Level 2 Events

передача синтетического события в данный узел

Синтаксис

```
boolean dispatchEvent(Event evt)
    throws EventException;
```

Аргументы

evt – передаваемый объект Event.

Возвращаемое значение

`false`, если в любой момент во время распространения события был вызван метод `preventDefault()` события *evt*, или `true` в противном случае.

Исключения

Этот метод генерирует исключение `EventException` со свойством `code`, равным `EventException.UNSPECIFIED_EVENT_TYPE_ERR`, если объект Event *evt* не был инициализирован или если его свойство `type` равно `null` или пустой строке.

Описание

Этот метод выполняет диспетчеризацию синтетического события, созданного с помощью `Document.createEvent()` и инициализированного с помощью метода инициализации, определенного интерфейсом Event или одним из его подынтерфейсов. Узел, для которого вызывается метод, становится целью события, но сначала событие распространяется вниз по дереву документа на стадии перехвата и затем, если свойство `bubbles` события равно `true`, «всплывает» вверх по дереву документа, после обработки в целевом узле события.

См. также

Document.createEvent(), Event.initEvent(), MouseEvent.initMouseEvent()

EventTarget.removeEventListener() DOM Level 2 Events

удаляет слушатель события

Синтаксис

```
void removeEventListener(String тип,
                         EventListener слушатель,
                         boolean использоватьПерехват);
```

Аргументы

тип

Тип события, для которого должен быть удален слушатель.

слушатель

Функция-слушатель события, которая должна быть удалена.

использоватьПерехват

`true`, если должен быть удален перехватывающий слушатель события; `false`, если должен быть удален обычный слушатель события.

Описание

Этот метод удаляет указанную функцию-слушатель события. Аргументы *тип* и *использоватьПерехват* должны быть такими же, как при соответствующем вызове `addEventListener()`. Если обработчик события, соответствующий указанным аргументам, не найден, этот метод не делает ничего.

Функция-слушатель события, удаленная с помощью этого метода, больше не будет вызываться для указанного *типа* события в данном узле. Причем даже если слушатель удаляется другим слушателем события, зарегистрированным для того же типа событий в том же узле.

HTMLAnchorElement DOM Level 1 HTML

гиперссылка или якорный элемент в HTML-документе **Node → Element → HTMLElement → HTMLAnchorElement**

Свойства

Этот интерфейс определяет свойства, перечисленные в следующей таблице, которые соответствуют атрибутам тега HTML `<a>`.

Свойство	Атрибут	Описание
`String accessKey`	`accesskey`	Клавиша быстрого вызова
`String charset`	`charset`	Кодировка целевого документа
`String coords`	`coords`	Используется внутри элемента `<map>`
`String href`	`href`	URL гиперссылки
`String hreflang`	`hreflang`	Язык документа, на который указывает ссылка
`String name`	`name`	Имя якорного элемента
`String rel`	`rel`	Тип ссылки
`String rev`	`rev`	Тип обратной ссылки
`String shape`	`shape`	Используется внутри элементов `<map>`

Свойство	Атрибут	Описание
long tabIndex	tabindex	Позиция ссылки в порядке перехода по клавише Tab
String target	target	Имя фрейма или окна, в котором должен быть показан целевой документ
String type	type	Тип содержимого целевого документа

Методы

blur()

Уводит фокус клавиатуры от ссылки.

focus()

Прокручивает документ так, чтобы якорь или ссылка стали видимыми, и присваивает ссылке клавиатурный фокус.

Описание

Этот интерфейс представляет тег <a> в документе HTML. href, name и target являются ключевыми свойствами, представляющими наиболее часто применяемые атрибуты тега.

Объекты HTMLAnchorElement могут быть получены из свойств типа HTMLCollection links и anchors интерфейса HTMLDocument.

Пример

```
// Получает целевой URL первой гиперссылки в документе
var url = document.links[0].href;
// Прокручивает документ, чтобы стал видимым якорный элемент "_bottom_"
document.anchors['_bottom_'].focus();
```

См. также

Объекты Link и Anchor в справочном разделе по клиентскому JavaScript

HTMLAnchorElement.blur() DOM Level 1 HTML

уводит клавиатурный фокус от гиперссылки

Синтаксис

```
void blur();
```

Описание

В веб-броузерах, допускающих получение клавиатурного фокуса гиперссылками, этот метод забирает клавиатурный фокус у гиперссылки.

HTMLAnchorElement.focus() DOM Level 1 HTML

делает ссылку или якорный элемент видимым и передает ему клавиатурный фокус

Синтаксис

```
void focus();
```

Описание

Этот метод прокручивает документ так, чтобы указанный якорный элемент или гиперссылка стали видимыми. Если элемент представляет собой гиперссылку, и броузер допускает получение клавиатурного фокуса гиперссылками, этот метод также передает клавиатурный фокус элементу.

HTMLBodyElement

DOM Level 1 HTML

тело (<body>) HTML-документа **Node → Element → HTMLElement → HTMLBodyElement**

Свойства

`deprecated String aLink`

Значение атрибута `alink`. Задает цвет «активных» ссылок, то есть цвет ссылки, когда клавиша мыши нажимается над ней, но до того как кнопка отпускается.

`deprecated String background`

Значение атрибута `background`. Задает URL изображения для фоновой текстуры документа.

`deprecated String bgColor`

Значение атрибута `bgcolor`. Задает цвет фона документа.

`deprecated String link`

Значение атрибута `link`. Задает обычный цвет гиперссылок, которые еще не открывались.

`deprecated String text`

Значение атрибута `text`. Задает цвет переднего плана (цвет текста) для документа.

`deprecated String vLink`

Значение атрибута `vlink`. Задает обычный цвет «посещенных» ссылок, которые уже открывались.

Описание

Интерфейс HTMLBodyElement представляет тег `<body>` документа. Во всех документах HTML имеется тег `<body>`, даже если он не присутствует явно в исходном тексте документа. HTMLBodyElement для документа можно получить через свойство `body` интерфейса HTMLDocument.

Свойства этого объекта задают цвета и изображения по умолчанию для документа. Несмотря на то что эти свойства представляют значения атрибутов `<body>`, DOM Level 0 сделал те же самые значения доступными через свойства объекта Document (с другими именами). Подробности см. в описании объекта Document в «Справочнике по клиентскому JavaScript» в этой книге.

Несмотря на то что эти свойства цвета и изображения больше подходят для интерфейса HTMLBodyElement, чем для объекта Document, обратите внимание, что все они признаны устаревшими, так как в HTML 4 признаны устаревшими представляемые ими атрибуты тега `<body>`. Предпочтительным способом задания цветов и изображений в документе является использование стилей CSS.

Пример

```
document.body.text = "#ff0000";        // Задает ярко-красный цвет текста
document.fgColor = "#ff0000";          // То же самое с использованием старого DOM Level 0 API
document.body.style.color= "#ff0000";  // То же самое с использованием стилей CSS
```

См. также

Объект Document в «Справочнике по клиентскому JavaScript»

HTMLCollection

DOM Level 1 HTML

массив элементов HTML, доступных по позиции или по имени

Свойства

`readonly unsigned long length`

Количество элементов в коллекции.

Методы

`item()`

Возвращает элемент коллекции, расположенный в указанной позиции. Можно не вызывать этот метод явно, а задать номер в квадратных скобках (как для массива).

`namedItem()`

Возвращает элемент из коллекции, имеющий указанное значение атрибута `id` или `name`, или `null`, если такого элемента нет. Можно также поместить имя элемента в квадратные скобки (как для массива), а не вызывать этот метод явно.

Описание

HTMLCollection – это коллекция элементов HTML с методами, позволяющими извлекать элементы либо по их позиции в документе, либо по их атрибуту `id` или `name`. В JavaScript объекты HTMLCollection ведут себя как доступные только для чтения массивы, и для индексирования HTMLCollection по номеру или по имени можно применять нотацию JavaScript с квадратными скобками, а не вызывать методы `item()` и `namedItem()`.

Многие свойства интерфейса HTMLDocument (стандартизующего объект Document DOM Level 0) – это объекты HTMLCollection, предоставляющие удобный доступ к таким элементам документа, как формы, изображения и ссылки. Объект HTMLCollection также предоставляет удобный способ обхода элементов формы HTML, строк таблиц HTML, ячеек в строке таблицы и областей в клиентскому изображению-карте.

Объекты HTMLCollection доступны только для чтения – им нельзя присваивать новые элементы даже при помощи нотации массивов JavaScript. Они «живые», то есть когда исходный документ изменяется, эти изменения сразу же становятся видимыми через объекты HTMLCollection.

Пример

```
var c = document.forms;          // Это HTMLCollection для элементов формы
var firstform = c[0];            // Он может использоваться как массив с числовым индексом
var lastform = c[c.length-1];    // Свойство length задает количество элементов
var address = c["address"];      // Он может использоваться как ассоциативный массив
var address = c.address;         // JavaScript допускает и такую нотацию
```

См. также

NodeList

Тип: HTMLDocument.anchors, HTMLDocument.applets, HTMLDocument.forms, HTMLDocument.images, HTMLDocument.links, HTMLFormElement.elements, HTMLMapElement.areas, HTMLSelectElement.options, HTMLTableElement.rows, HTMLTableElement.tBodies, HTMLTableRowElement.cells, HTMLTableSectionElement.rows

HTMLCollection.item() DOM Level 1 HTML

получает элемент по его позиции

Синтаксис

`Node item(unsigned long` *`индекс`*`);`

Аргументы

индекс

Позиция элемента, который должен быть возвращен. Элементы расположены в HTMLCollection в том же порядке, что и в исходном тексте документа.

Возвращаемое значение

Элемент в позиции *индекс* или `null`, если *индекс* меньше нуля либо больше или равен свойству `length`.

Описание

Метод `item()` возвращает элемент с указанным номером из HTMLCollection. В JavaScript проще рассматривать HTMLCollection как массив и обращаться к нему с помощью нотации массива.

Пример

```
var c = document.images;  // Это HTMLCollection
var img0 = c.item(0);     // Можно использовать метод item() так
var img1 = c[1];          // Но эта нотация проще и применяется чаще
```

См. также

NodeList.item()

HTMLCollection.namedItem() DOM Level 1 HTML

получает элемент по имени

Синтаксис

`Node namedItem(String` *`имя`*`);`

Аргументы

имя

Имя возвращаемого элемента.

Возвращаемое значение

Элемент с указанным *именем*, или `null`, если ни один из элементов HTMLCollection не имеет такого имени.

Описание

Этот метод находит и возвращает элемент из HTMLCollection, имеющий указанное имя. Если какой-либо элемент имеет атрибут id, значение которого равно указанному имени, то возвращается этот элемент. Если такой элемент не найден, возвращается элемент, атрибут name которого равен указанному значению. Если такого элемента не существует, namedItem() возвращает null.

Обратите внимание, что атрибут id может быть задан для любого элемента HTML, но только определенные элементы HTML, такие как формы, элементы формы, изображения и якорные элементы, могут иметь атрибут name.

В JavaScript проще рассматривать HTMLCollection как ассоциативный массив и задававать *имя* между квадратными скобками, используя нотацию массива.

Пример

```
var forms = document.forms;                  // Коллекция HTMLCollection форм
var address = forms.namedItem("address");    // Находит <form name="address">
var payment = forms["payment"]               // Более простой синтаксис: находит тег
                                             // <form name="payment">
var login = forms.login;                     // Тоже работает: находит тег <form name="login">
```

HTMLDocument

DOM Level 1 HTML

вершина дерева HTML-документа **Node → Document → HTMLDocument**

Свойства

readonly HTMLCollection anchors

Массив (HTMLCollection) всех якорных элементов в документе. Из соображений совместимости с DOM Level 0 этот массив содержит только элементы <a>, для которых указан атрибут name; он не включает якоря, созданные с атрибутом id.

readonly HTMLCollection applets

Массив (HTMLCollection) всех апплетов в документе. Он включает апплеты, определенные с помощью тега <object> и всех тегов <applet>.

HTMLElement body

Вспомогательное свойство, которое ссылается на объект HTMLBodyElement, представляющий тег <body> этого документа. Для документов, определяющих наборы фреймов, это свойство ссылается на самый внешний тег <frameset>.

String cookie

Позволяет получать и устанавливать cookie для этого документа. См. статью «Document.cookie» в «Справочнике по клиентскому JavaScript».

readonly String domain

Имя домена сервера, откуда был загружен документ, или null, если он отсутствует. Сравните с доступным для чтения и записи свойством Document.domain в «Справочнике по клиентскому JavaScript».

readonly HTMLCollection forms

Массив (HTMLCollection) всех объектов HTMLFormElement в документе.

`readonly HTMLCollection images`

Массив (HTMLCollection) всех тегов `<img>` в документе. Обратите внимание, что для обеспечения совместимости с DOM Level 0 изображения, определенные с помощью тега `<object>`, не включаются в эту коллекцию.

`readonly HTMLCollection links`

Массив (HTMLCollection) всех гиперссылок в документе. Он включает все теги `<a>`, имеющие атрибут `href`, и все теги `<area>`.

`readonly String referrer`

URL документа, ссылающегося на данный документ, или `null`, если этот документ был открыт не через гиперссылку.

`String title`

Содержимое тега `<title>` для этого документа.

`readonly String URL`

URL документа.

Методы

`close()`

Закрывает поток документа, открытый с помощью метода `open()`, заставляя вывести любые буферизованные данные.

`getElementById()`

Возвращает элемент с указанным `id`. В DOM Level 2 этот метод наследуется от интерфейса Document.

`getElementsByName()`

Возвращает массив узлов (NodeList) всех элементов в документе, значение атрибута `name` которых равно указанному.

`open()`

Открывает поток, в который может быть записано содержимое нового документа. Обратите внимание, что этот метод удаляет любое текущее содержимое документа.

`write()`

Дописывает строку текста HTML в открытый документ.

`writeln()`

Дописывает строку текста HTML и символ перевода строки в открытый документ.

Описание

Этот интерфейс расширяет интерфейс Document и определяет специфические для HTML свойства и методы, обеспечивающие совместимость с объектом Document DOM Level 0 (см. описание объекта Document в «Справочнике по клиентскому JavaScript»). Обратите внимание, что в HTMLDocument имеются не все свойства объекта Document Level 0. Свойства, задающие цвета и фоновые изображения документа, были переименованы и перемещены в HTMLBodyElement.

И наконец, заметьте, что в DOM Level 1 HTMLDocument определяет метод с именем `getElementById()`. В DOM Level 2 этот метод был перемещен в интерфейс Document, и теперь наследуется интерфейсом HTMLDocument, а не определяется в нем. Подробности см. в статье «Document.getElementById()» данного справочника.

См. также

Document.getElementById(), HTMLBodyElement; Document object в «Справочнике по клиентскому JavaScript»

Возвращается из: HTMLDOMImplementation.createHTMLDocument()

HTMLDocument.close() DOM Level 1 HTML

закрывает открытый документ и выводит его

Синтаксис

```
void close();
```

Описание

Этот метод закрывает поток документа, который был открыт с помощью метода `open()`, и заставляет вывести все буферизованные данные. Подробности см. в статье «Document.close()» в «Справочнике по клиентскому JavaScript».

См. также

HTMLDocument.open(); Document.close() в «Справочнике по клиентскому JavaScript»

HTMLDocument.getElementById()

См. Document.getElementById()

HTMLDocument.getElementsByName() DOM Level 1 HTML

находит элементы с указанным атрибутом name

Синтаксис

```
Node[] getElementsByName(String имяЭлемента);
```

Аргументы

имяЭлемента

Желаемое значение атрибута `name`.

Возвращаемое значение

Массив (фактически NodeList) элементов, имеющих атрибут `name`, равный указанному значению. Если такие элементы не найдены, возвращаемый NodeList пуст, и его длина равна 0.

Описание

Этот метод ищет в дереве HTML-документа узлы Element, в которых атрибут `name` равен указанному значению, и возвращает NodeList (который можно рассматривать как массив), содержащий все найденные элементы. Если элементы не найдены, возвращается NodeList со свойством `length`, равным 0.

Не путайте этот метод с методом `Document.getElementById()`, который ищет одиночный узел Element на основе уникального значения атрибута `id`, или с методом `Document.getElementsByTagName()`, возвращающим NodeList элементов с указанным именем тега.

См. также

Document.getElementById(), Document.getElementsByTagName()

HTMLDocument.open() DOM Level 1 HTML

начинает новый документ, стирая текущий

Синтаксис

```
void open();
```

Описание

Этот метод стирает текущий документ HTML и начинает новый, в который можно писать с помощью методов `write()` и `writeln()`. После вызова метода `open()` для открытия нового документа и метода `write()` для задания содержимого документа необходимо вызвать метод `close()` для завершения документа и вывода его содержимого.

Этот метод не должен вызываться сценарием или обработчиком событий, являющимся частью переписываемого документа, так как сам сценарий или обработчик также будет переписан.

См. статью «Document.open()» в «Справочнике по клиентскому JavaScript», но обратите внимание, что эта стандартизованная версия метода может применяться только для создания новых HTML-документов и не принимает дополнительный аргумент *mimetype*, который поддерживается в версии Level 0.

Пример

```
var w = window.open("");            // Открывает новое окно
var d = w.document;                 // Получает его объект HTMLDocument
d.open();                           // Открывает документ для записи
d.write("<h1>Hello world</h1>");    // Выводит в документ некоторый HTML-код
d.close();                          // Завершает и отображает документ
```

См. также

HTMLDocument.close(), HTMLDocument.write(); Document.open() в «Справочнике по клиентскому JavaScript»

HTMLDocument.write() DOM Level 1 HTML

дописывает текст HTML в открытый документ

Синтаксис

```
void write(String текст);
```

Аргументы

текст

Дописываемый к документу текст HTML.

Описание

Этот метод дописывает указанный текст HTML к документу, который был открыт с помощью метода `open()` и еще не был закрыт с помощью `close()`.

См. статью «Document.open()» в «Справочнике по клиентскому JavaScript», но обратите внимание, что эта стандартизованная версия метода DOM Level 0 принимает только один строковый аргумент, а не произвольное количество аргументов.

См. также

HTMLDocument.open(); Document.write() в «Справочнике по клиентскому JavaScript»

HTMLDocument.writeln() DOM Level 1 HTML

дописывает текст HTML и символ перевода строки в открытый документ

Синтаксис

```
void writeln(String текст);
```

Аргументы

text

Текст HTML, дописываемый к документу.

Описание

Этот метод похож на `HTMLDocument.write()` за исключением того, что за дописываемым текстом следует символ новой строки, который может быть полезен, например, при формировании содержимого тега `<pre>`.

См. также

Document.writeln() в «Справочнике по клиентскому JavaScript»

HTMLDOMImplementation

См. DOMImplementation

HTMLElement DOM Level 1 HTML

базовый интерфейс для всех элементов HTML **Node → Element → HTMLElement**

Также реализует

`ElementCSSInlineStyle`

Если реализация поддерживает таблицы стилей CSS, то и все объекты, реализующие этот интерфейс, также реализуют интерфейс ElementCSSInlineStyle. Так как поддержка CSS распространена в веб-броузерах, свойство `style`, определенное этим интерфейсом, для удобства включено в эту статью.

Подынтерфейсы

HTMLAnchorElement	HTMLAppletElement	HTMLAreaElement
HTMLBRElement	HTMLBaseElement	HTMLBaseFontElement
HTMLBodyElement	HTMLButtonElement	HTMLDListElement
HTMLDirectoryElement	HTMLDivElement	HTMLFieldSetElement
HTMLFontElement	HTMLFormElement	HTMLFrameElement
HTMLFrameSetElement	HTMLHRElement	HTMLHeadElement
HTMLHeadingElement	HTMLHtmlElement	HTMLIFrameElement
HTMLImageElement	HTMLInputElement	HTMLIsIndexElement
HTMLLIElement	HTMLLabelElement	HTMLLegendElement
HTMLLinkElement	HTMLMapElement	HTMLMenuElement
HTMLMetaElement	HTMLModElement	HTMLOListElement
HTMLParagraphElement	HTMLParamElement	HTMLPreElement

HTMLQuoteElement	HTMLScriptElement	HTMLSelectElement
HTMLStyleElement	HTMLTableCaptionElement	HTMLTableCellElement
HTMLTableColElement	HTMLTableElement	HTMLTableRowElement
HTMLTableSectionElement	HTMLTextAreaElement	HTMLTitleElement
HTMLUListElement		

Свойства

`readonly CSS2Properties style`

Значение атрибута `style`, задающее встроенные стили CSS для этого элемента. Это свойство на самом деле определено не непосредственно в интерфейсе HTMLElement, а в интерфейсе ElementCSSInlineStyle. Если броузер поддерживает таблицы стилей CSS, все объекты HTMLElement в нем реализуют ElementCSSInlineStyle и определяют это свойство `style`. Значение этого свойства представляет собой объект, реализующий интерфейс CSSStyleDeclaration и (более часто используемый) интерфейс CSS2Properties.

`String className`

Значение атрибута `class` элемента, задающее имя класса CSS. Обратите внимание, у этого свойства нет имени «class», так как это имя в JavaScript зарезервировано.

`String dir`

Значение атрибута `dir` элемента, задающее направление текста в документе.

`String id`

Значение атрибута `id`. Никакие два элемента в одном документе не должны иметь одинаковые значения `id`.

`String lang`

Значение атрибута `lang`, задающее код языка для документа.

`String title`

Значение атрибута `title`, задающее описательный текст для отображения в качестве «подсказки» для элемента.

Описание

Этот интерфейс определяет свойства, которые представляют атрибуты, общие для всех элементов HTML. Все элементы HTML реализуют этот интерфейс, и большинство из них реализует специфический для тега подынтерфейс, который определяет свойства для каждого из атрибутов этого тега. В дополнение к свойствам, перечисленным здесь, на справочной странице «HTMLElement» в «Справочнике по клиентскому JavaScript» приведен список свойств-обработчиков событий DOM Level 0, которые поддерживаются всеми элементами HTML в документе.

Некоторые теги HTML не допускают никаких атрибутов, за исключением универсальных, допустимых для всех тегов HTML и представляемых свойствами HTMLElement. Эти теги не имеют собственного специального подынтерфейса, и элементы этого типа в дереве документа представлены объектом HTMLElement. Теги, не имеющие собственного специального подынтерфейса, приведены в следующем списке:

`<abbr>`	`<acronym>`	`<address>`	`<b>`
`<bdo>`	`<big>`	`<center>`	`<cite>`

<code>	<dd>	<dfn>	<dt>
	<i>	<kbd>	<noframes>
<noscript>	<s>	<samp>	<small>
	<strike>		<sub>
<sup	<tt>	<u>	<var>

Как было показано в предыдущем разделе «Подынтерфейсы», для многих тегов HTML *существуют* специальные подынтерфейсы. Обычно для тега с именем *T* имеется специальный интерфейс с именем HTML*T*Element. Например, тег <head> представляется интерфейсом HTMLHeadElement. В некоторых случаях несколько связанных тегов разделяют один интерфейс, как в случае с тегами от <h1> до <h6>, которые представлены интерфейсом HTMLHeadingElement.

Большинство этих специфических интерфейсов не делают ничего, кроме определения свойств JavaScript для всех атрибутов тега HTML. Свойства JavaScript имеют те же имена, что и атрибуты, и состоят из символов нижнего регистра (например, id) или, когда имя атрибута состоит из нескольких слов, смешанного регистра (например, className). Если имя атрибута HTML является зарезервированным словом Java или JavaScript, то оно слегка меняется. Например, атрибут class всех тегов HTML становится свойством className интерфейса HTMLElement, так как class – это зарезервированное слово. Аналогично, атрибут for тегов <label> и <script> становится свойством htmlFor интерфейсов HTMLLabelElement и HTMLScriptElement, так как for – это зарезервированное слово. Назначение этих свойств непосредственно соответствует атрибутам HTML, определенным в спецификации HTML, и описание каждого из них выходит за рамки этой книги.

В следующей таблице перечислены все теги HTML, для которых имеется соответствующий подынтерфейс HTMLElement. Для каждого тега в таблице указано имя интерфейса и имена определенных в нем свойств и методов. Все свойства являются доступными для чтения и записи строками, если не указано другое. Для свойств, не являющихся строками, тип свойства указывается в квадратных скобках перед именем свойства. Эти интерфейсы и их свойства столь непосредственно соответствуют элементам HTML и атрибутам, что для большинства интерфейсов в этой книге нет отдельных справочных страниц, и подробную информацию об этих интерфейсах следует искать в литературе по HTML. Исключение составляют интерфейсы, в которых определены методы, и интерфейсы, которые представляют некоторые особенно важные теги, такие как тег <body>. Эти интерфейсы помечены в таблице знаком *, и сведения о них можно найти в данном справочнике.

Тег HTML	Интерфейс, свойства и методы DOM
все теги	**HTMLElement:**[a] id, title, lang, dir, className
<a>	**HTMLAnchorElement**[a]**:** accessKey, charset, coords, href, hreflang, name, rel, rev, shape, [long] tabIndex, target, type, blur(), focus()
<applet>	**HTMLAppletElement**[b]**:** align[b], alt[b], archive[b], code[b], codeBase[b], height[b], hspace[b], name[b], object[b], vspace[b], width[b]
<area>	**HTMLAreaElement:** accessKey, alt, coords, href, [boolean] noHref, shape, [long] tabIndex, target
<base>	**HTMLBaseElement:** href, target

Тег HTML	Интерфейс, свойства и методы DOM
<basefont>	**HTMLBaseFontElement**[b]**:** color[b], face[b], size[b]
<blockquote>, <q>	**HTMLQuoteElement:** cite
<body>	**HTMLBodyElement**[a]**:** aLink[b], background[b], bgColor[b], link[b], text[b], vLink[b]
 	HTMLBRElement: clear[b]
<button>	**HTMLButtonElement:** [readonly HTMLFormElement] form, accessKey, [boolean] disabled, name, [long] tabIndex, [readonly] type, value
<caption>	**HTMLTableCaptionElement**[a]**:** align[b]
<col>, <colgroup>	**HTMLTableColElement**[a]**:** align, ch, chOff, [long] span, vAlign, width
<del>, <ins>	**HTMLModElement:** cite, dateTime
<dir>	**HTMLDirectoryElement**[b]**:** [boolean] compact[b]
	HTMLDivElement: align[b]
<dl>	**HTMLDListElement:** [boolean] compact[b]
<fieldset>	**HTMLFieldSetElement:** [readonly HTMLFormElement] form
<font>	**HTMLFontElement**[b]**:** color[b], face[b], size[b]
<form>	**HTMLFormElement**[a]**:** [readonly HTMLCollection] elements, [readonly long] length, name, acceptCharset, action, enctype, method, target, submit(), reset()
<frame>	**HTMLFrameElement:** frameBorder, longDesc, marginHeight, marginWidth, name, [boolean] noResize, scrolling, src, [readonly Document] contentDocument[c]
<frameset>	**HTMLFrameSetElement:** cols, rows
<h1>, <h2>, <h3>, <h4>, <h5>, <h6>	**HTMLHeadingElement:** align[b]
<head>	**HTMLHeadElement:** profile
<hr>	**HTMLHRElement:** align[b], [boolean] noShade[b], size[b], width[b]
<html>	**HTMLHtmlElement:** version[b]
<iframe>	**HTMLIFrameElement:** align[b], frameBorder, height, longDesc, marginHeight, marginWidth, name, scrolling, src, width, [readonly Document] contentDocument[c]
<img>	**HTMLImageElement:** align[b], alt, [long] border[b], [long] height, [long] hspace[b], [boolean] isMap, longDesc, name, src, useMap, [long] vspace[b], [long] width
<input>	**HTMLInputElement**[a]**:** defaultValue, [boolean] defaultChecked, [readonly HTMLFormElement] form, accept, accessKey, align[b], alt, [boolean] checked, [boolean] disabled, [long] maxLength, name, [boolean] readOnly, size, src, [long] tabIndex, type, useMap, value, blur(), focus(), select(), click()
<ins>	См. <del>
<isindex>	**HTMLIsIndexElement**[b]**:** [readonly HTMLFormElement] form, prompt[b]

(продолжение)

Тег HTML	Интерфейс, свойства и методы DOM
<label>	**HTMLLabelElement:** [readonly HTMLFormElement] form, accessKey, htmlFor
<legend>	**HTMLLegendElement:** [readonly HTMLFormElement] form, accessKey, align[b]
	HTMLLIEle ment: type[b], [long] value[b]
<link>	**HTMLLinkElement:** [boolean] disabled, charset, href, hreflang, media, rel, rev, target, type
<map>	**HTMLMapElement:** [readonly HTMLCollection of HTMLAreaElement] areas, name
<menu>	**HTMLMenuElement[b]:** [boolean] compact[b]
<meta>	**HTMLMetaElement:** content, httpEquiv, name, scheme
<object>	**HTMLObjectElement:** code, align[b], archive, border[b], codeBase, codeType, data, [boolean] declare, height, hspace[b], name, standby, [long] tabIndex, type, useMap, vspace[b], width, [readonly Document] contentDocument[c]
	HTMLOListElement: [boolean] compact[b], [long] start[b], type[b]
<optgroup>	**HTMLOptGroupElement:** [boolean] disabled, label
<option>	**HTMLOptionElement[a]:** [readonly HTMLFormElement] form, [boolean] defaultSelected, [readonly] text, [readonly long] index, [boolean] disabled, label, [boolean] selected, value
<p>	**HTMLParagraphElement:** align[b]
<param>	**HTMLParamElement:** name, type, value, valueType
<pre>	**HTMLPreElement:** [long] width[b]
<q>	**См.** <blockquote>
<script>	**HTMLScriptElement:** text, htmlFor, event, charset, [boolean] defer, src, type
<select>	**HTMLSelectElement[a]:** [readonly] type, [long] selectedIndex, value, [readonly long] length, [readonly HTMLFormElement] form, [readonly HTMLCollection of HTMLOptionElement] options, [boolean] disabled, [boolean] multiple, name, [long] size, [long] tabIndex, add(), remove(), blur(), focus()
<style>	**HTMLStyleElement:** [boolean] disabled, media, type
<table>	**HTMLTableElement[a]:** [HTMLTableCaptionElement] caption, [HTMLTableSectionElement] tHead, [HTMLTableSectionElement] tFoot, [readonly HTMLCollection of HTMLTableRowElement] rows, [readonly HTMLCollection of HTMLTableSectionElement] tBodies, align[b], bgColor[b], border, cellPadding, cellSpacing, frame, rules, summary, width, createTHead(), deleteTHead(), createTFoot(), deleteTFoot(), createCaption(), deleteCaption(), insertRow(), deleteRow()

Тег HTML	Интерфейс, свойства и методы DOM
`<tbody>`, `<tfoot>`, `<thead>`	**HTMLTableSectionElement**[a]: `align`, `ch`, `chOff`, `vAlign`, `[readonly HTMLCollection of HTMLTableRowElement] rows`, `insertRow()`, `deleteRow()`
`<td>`, `<th>`	**HTMLTableCellElement**[a]: `[readonly long] cellIndex`, `abbr`, `align`, `axis`, `bgColor`[b], `ch`, `chOff`, `[long] colSpan`, `headers`, `height`[b], `[boolean] noWrap`[b], `[long] rowSpan`, `scope`, `vAlign`, `width`[b]
`<textarea>`	**HTMLTextAreaElement**[a]: `defaultValue`, `[readonly HTMLFormElement] form`, `accessKey`, `[long] cols`, `[boolean] disabled`, `name`, `[boolean] readOnly`, `[long] rows`, `[long] tabIndex`, `[readonly] type`, `value`, `blur()`, `focus()`, `select()`
`<tfoot>`	См. `<tbody>`
`<th>`	См. `<td>`
`<thead>`	См. `<tbody>`
`<title>`	**HTMLTitleElement**: `text`
`<tr>`	**HTMLTableRowElement**[a]: `[readonly long] rowIndex`, `[readonly long] sectionRowIndex`, `[readonly HTMLCollection of HTMLTableCellElement] cells`, `align`, `bgColor`[b], `ch`, `chOff`, `vAlign`, `insertCell()`, `deleteCell()`
`<ul>`	**HTMLUListElement**: `[boolean] compact`[b], `type`[b]

[a] Обозначает интерфейсы, описанные в этой книге.

[b] Обозначает элементы и атрибуты, признанные устаревшими.

[c] Обозначает атрибуты, добавленные в рабочий проект HTML DOM Level 2.

См. также

HTMLElement в «Справочнике по клиентскому JavaScript»

Тип: HTMLDocument.body

Передается в: HTMLSelectElement.add()

Возвращается из: HTMLTableElement.createCaption(), HTMLTableElement.createTFoot(), HTMLTableElement.createTHead(), HTMLTableElement.insertRow(), HTMLTableRowElement.insertCell(), HTMLTableSectionElement.insertRow()

HTMLFormElement

DOM Level 1 HTML

форма в HTML-документе **Node → Element → HTMLElement → HTMLFormElement**

Свойства

`readonly HTMLCollection elements`

Массив (HTMLCollection) всех элементов в форме.

`readonly long length`

Количество элементов в форме. Это значение идентично `elements.length`.

В дополнение к приведенным выше свойствам, HTMLFormElement также определяет свойства, перечисленные в следующей таблице, которые непосредственно соответствуют атрибутам HTML.

Свойство	Атрибут	Описание
String acceptCharset	acceptcharset	Наборы символов, которые может принимать сервер
String action	action	URL обработчика формы
String enctype	enctype	Кодировка формы
String method	method	HTTP-метод, используемый для подтверждения формы
String name	name	Имя формы
String target	target	Имя фрейма или окна для результатов подтверждения формы

Методы

reset()

Сбрасывает все элементы формы к их значениям по умолчанию.

submit()

Подтверждает форму.

Описание

Этот интерфейс представляет элемент <form> в HTML-документе. Свойство elements — это коллекция HTMLCollection, предоставляющая удобный доступ ко всем элементам формы. Методы submit() и reset() позволяют подтвердить или сбросить форму из программы JavaScript. Подробности см. в описании объекта Form в «Справочнике по клиентскому JavaScript».

См. также

Объект Form в «Справочнике по клиентскому JavaScript»

Тип: HTMLButtonElement.form, HTMLFieldSetElement.form, HTMLInputElement.form, HTMLIsIndexElement.form, HTMLLabelElement.form, HTMLLegendElement.form, HTMLObjectElement.form, HTMLOptionElement.form, HTMLSelectElement.form, HTMLTextAreaElement.form

HTMLFormElement.reset() DOM Level 1 HTML

сбрасывает элементы формы к их значениям по умолчанию

Синтаксис

```
void reset();
```

Описание

Этот метод сбрасывает все элементы формы к их значению по умолчанию. Результаты вызова этого метода схожи с результатами щелчка по кнопке Reset, за исключением того, что не вызывается обработчик события onreset формы.

См. также

Form.reset() в «Справочнике по клиентскому JavaScript»

HTMLFormElement.submit() DOM Level 1 HTML

передает данные формы

Синтаксис

```
void submit();
```

Описание

Этот метод отправляет значения элементов формы в обработчик формы, заданный в свойстве `action`. Передача данных формы выполняется так же, как это происходит при щелчке по кнопке Submit, за исключением того, что не вызывается обработчик события `onsubmit` формы.

См. также

Form.submit() в «Справочнике по клиентскому JavaScript»

HTMLInputElement DOM Level 1 HTML

элемент ввода в форме HTML **Node → Element → HTMLElement → HTMLInputElement**

Свойства

`String accept`

Список MIME-типов (через запятую), задающих типы файлов, которые могут загружаться в том случае, если это элемент FileUpload. Соответствует атрибуту `accept`.

`String accessKey`

Клавиша быстрого доступа (должна быть одиночным символом), которую броузер может использовать для передачи клавиатурного фокуса этому элементу ввода. Соответствует атрибуту `accesskey`.

`deprecated String align`

Вертикальное выравнивание этого элемента относительно окружающего текста, левое или правое обтекание элемента.

`String alt`

Альтернативный текст, отображаемый броузерами, которые не могут отобразить этот элемент ввода. Особенно полезен, когда `type` равно `image`. Соответствует атрибуту `alt`.

`boolean checked`

Для элементов Radio и Checkbox задает, отмечен ли элемент. Установка этого свойства изменяет внешний вид элемента ввода. Соответствует атрибуту `checked`.

`boolean defaultChecked`

В этом свойстве содержится исходное значение атрибута `checked` для элементов Radio и Checkbox, как оно указано в исходном тексте документа. Когда форма сбрасывается, свойство `checked` возвращается к значению этого свойства. Изменение значения этого свойства изменяет значение свойства `checked` и текущего состояния отметки элемента.

`String defaultValue`

Содержит исходное значение для элементов Text, Password и FileUpload, отображаемое элементом. Когда форма сбрасывается, элемент восстанавливается к это-

му значению. Изменение значения этого свойства также изменяет значение свойства value и текущее отображаемое значение.

boolean disabled

Если свойство равно true, элемент ввода отключен и недоступен для пользовательского ввода. Соответствует атрибуту disabled.

readonly HTMLFormElement form

Объект HTMLFormElement, представляющий элемент <form>, в котором содержится этот элемент ввода, или null, если элемент ввода не находится внутри формы.

long maxLength

Задает максимальное количество символов для элемента Text или Password, которое пользователю будет разрешено ввести. Обратите внимание, что это не то же самое, что свойство size. Соответствует атрибуту maxlength.

String name

Имя элемента ввода, заданное в атрибуте name.

boolean readOnly

Если свойство равно true и это элемент Text или Password, пользователю не разрешается вводить текст в элемент. Соответствует атрибуту readonly.

String size

Задает ширину элемента в символах для элементов Text и Password. Соответствует атрибуту size. См. также maxLength.

String src

Задает URL выводимого изображения для элементов ввода, для которых type=image. Соответствует атрибуту src.

long tabIndex

Позиция этого элемента ввода в порядке обхода по нажатию клавиши Tab. Соответствует атрибуту tabindex.

String type

Тип элемента ввода. Различные типы и их значения приведены в таблице в разделе «Описание». Соответствует атрибуту type.

String useMap

Для элементов, у которых type равно image, задает имя элемента <map>, предоставляющее клиентскую карту изображения для элемента.

String value

Значение, которое передается серверному сценарию при подтверждении формы. Для элементов Text, Password и FileUpload это свойство представляет собой текст, содержащийся в элементе ввода. Для элементов Button, Submit и Reset это текст, появляющийся в кнопке. По соображениям безопасности свойство value элементов FileUpload может быть доступно только для чтения. Аналогично, значение, возвращаемое этим свойством для элементов Password, может не содержать реально введенных пользователем данных.

Методы

`blur()`

Забирает клавиатурный фокус от элемента.

`click()`

Если этот элемент ввода – кнопка Button, Checkbox, Radio, Submit или Reset, этот метод имитирует нажатие кнопки мыши на элементе.

`focus()`

Передает клавиатурный фокус в этот элемент ввода.

`select()`

Если данный элемент ввода – элемент Text, Password или FileUpload, этот метод выделяет текст, который отображается элементом. Во многих броузерах это означает, что когда пользователь затем введет символ, выбранный текст будет удален и заменен на новый введенный символ.

Описание

Этот интерфейс представляет элемент `<input>`, определяющий элемент ввода в HTML (обычно в форме HTML). HTMLInputElement может представлять различные типы элементов ввода, в зависимости от значения свойства `type`. Допустимые значения этого свойства и их смысл показаны в следующей таблице.

Type	Тип элемента ввода
`button`	Кнопка
`checkbox`	Флажок
`file`	Элемент FileUpload
`hidden`	Скрытый элемент
`image`	Графическая кнопка
`password`	Маскированное текстовое поле ввода для паролей
`radio`	Переключатель
`reset`	Кнопка сброса
`text` (по умолчанию)	Однострочное текстовое поле ввода
`submit`	Кнопка подтверждения

Дополнительную информацию о формах HTML и элементах формы см. в главе 15. Обратите также внимание, что для каждого типа элементов формы имеется собственная справочная страница в «Справочнике по клиентскому JavaScript» в этой книге.

См. также

HTMLFormElement, HTMLOptionElement, HTMLSelectElement, HTMLTextAreaElement; главу 15; объект Input и его подклассы (Button, Checkbox, FileUpload, Hidden, Password, Radio, Reset, Submit и Text) в «Справочнике по клиентскому JavaScript»

HTMLInputElement.blur() DOM Level 1 HTML

забирает клавиатурный фокус у этого элемента

Синтаксис

```
void blur();
```

Описание

Этот метод забирает клавиатурный фокус у данного элемента формы.

HTMLInputElement.click() DOM Level 1 HTML

имитирует нажатие кнопки мыши на элементе формы

Синтаксис

```
void click();
```

Описание

Этот метод имитирует щелчок по элементу формы Button, Checkbox, Radio, Reset или Submit. Он не вызывает обработчик события `onclick` для элемента ввода.

При вызове для элементов Button он может производить (или не производить) визуальный эффект нажатия кнопки, но больше он ни на что не влияет, так как не вызывает обработчик события `onclick` для кнопки. Для элементов Checkbox он переключает состояние свойства `checked`. Он делает неотмеченные элементы Radio отмеченными, но не изменяет отмеченные элементы. При вызове для элементов Reset и Submit метод `click()` приводит к сбросу или передаче формы.

HTMLInputElement.focus() DOM Level 1 HTML

передает клавиатурный фокус этому элементу

Синтаксис

```
void focus();
```

Описание

Этот метод передает клавиатурный фокус данному элементу, чтобы пользователь мог взаимодействовать с элементом без предварительного щелчка по элементу.

HTMLInputElement.select() DOM Level 1 HTML

выделяет содержимое элемента Text

Синтаксис

```
void select();
```

Описание

Этот метод отмечает любой текст, отображаемый в элементах Text, Password и FileUpload. В большинстве броузеров это означает, что следующее нажатие клавиши вызовет замену текущего текста, а не добавление к нему.

HTMLOptionElement

DOM Level 1 HTML

опция в форме HTML **Node → Element → HTMLElement → HTMLOptionElement**

Свойства

`boolean defaultSelected`

Начальное значение атрибута `selected` элемента `<option>`. Если форма сбрасывается, свойство `selected` устанавливается равным значению этого свойства. Установка этого свойства также устанавливает значение свойства `selected`.

`boolean disabled`

Если равно `true`, эта опция отключена, и пользователю не разрешено ее выбирать. Соответствует атрибуту `disabled`.

`readonly HTMLFormElement form`

Ссылка на элемент `<form>`, содержащий данный элемент.

`readonly long index`

Позиция этого элемента `<option>` внутри содержащего его элемента `<select>`.

`String label`

Текст, который должен отображаться для этой опции. Соответствует атрибуту `label`. Если это свойство не указано, вместо него используется текстовое содержимое элемента `<option>`.

`boolean selected`

Текущее состояние этой опции: если `true`, опция выбрана. Начальное значение этого свойства берется из атрибута `selected`.

`readonly String text`

Текст, содержащийся в элементе `<option>` и появляющийся в виде метки опции.

`String value`

Значение, отправляемое с формой, если эта опция выбрана при подтверждении формы. Соответствует атрибуту `value`.

Описание

Этот интерфейс описывает элемент `<option>` внутри элемента `<select>`.

См. также

HTMLFormElement, HTMLInputElement, HTMLSelectElement; объекты Option и Select в «Справочнике по клиентскому JavaScript; главу 15

HTMLSelectElement

DOM Level 1 HTML

элемент <select> в форме HTML **Node → Element → HTMLElement → HTMLSelectElement**

Свойства

`boolean disabled`

Если это свойство равно `true`, элемент `<select>` отключен и пользователь не может с ним взаимодействовать. Соответствует атрибуту `disabled`.

`readonly HTMLFormElement form`

Элемент `<form>`, содержащий данный элемент.

`readonly long length`

Количество элементов `<option>`, содержащихся в этом элементе `<select>`. То же самое, что `options.length`.

`boolean multiple`

Если это свойство равно `true`, элемент `<select>` допускает выбор нескольких опций. В противном случае опции являются взаимоисключающими, и одновременно может быть выбрана только одна из них. Соответствует атрибуту `multiple`.

`String name`

Имя этого элемента формы. Соответствует атрибуту `name`.

`readonly HTMLCollection options`

Массив (HTMLCollection) объектов HTMLOptionElement, представляющих элементы `<option>`, которые содержатся в данном элементе `<select>`, в порядке их расположения в исходном тексте.

`long selectedIndex`

Позиция выбранной опции в массиве `options`. Если ни одна из опций не выбрана, это свойство равно –1. Если выбрано несколько опций, это свойство возвращает индекс первой выбранной опции.

`long size`

Количество одновременно отображаемых опций. Если оно равно 1, элемент `<select>` обычно отображается в виде выпадающего меню или списка. Если оно больше 1, `<select>` обычно отображается в виде окна списка фиксированного размера, в случае необходимости с полосой прокрутки. Соответствует атрибуту `size`.

`long tabIndex`

Позиция этого элемента в порядке обхода по нажатию клавиши Tab. Соответствует индексу `tabindex`.

`readonly String type`

Если `multiple` равно `true`, это свойство имеет значение «select-multiple». В противном случае оно равно «select-one».

Методы

`add()`

Вставляет новый HTMLOptionElement в массив `options`, добавляя его в конец массива либо перед другой указанной опцией.

`blur()`

Забирает клавиатурный фокус.

`focus()`

Передает клавиатурный фокус этому элементу.

`remove()`

Удаляет элемент `<option>` из указанной позиции.

Описание

Этот интерфейс представляет элемент `<select>` в форме HTML. Свойство `options` предоставляет удобный способ установки содержащихся в нем элементов `<option>`, а методы `add()` и `remove()` предоставляют простой способ изменения набора опций.

См. также

HTMLFormElement, HTMLOptionElement; объекты Option и Select в «Справочнике по клиентскому JavaScript»; главу 15

HTMLSelectElement.add() DOM Level 1 HTML

вставляет элемент <option>

Синтаксис

```
void add(HTMLElement элемент,
         HTMLElement перед)
    throws DOMException;
```

Аргументы

элемент

Добавляемый HTMLOptionElement.

перед

Элемент массива `options`, перед которым должен быть вставлен новый *элемент*. Если этот аргумент равен `null`, *элемент* добавляется в конец массива `options`.

Исключения

Этот метод генерирует исключение DOMException с кодом `NOT_FOUND_ERR`, если аргумент *перед* задает объект, не являющийся членом массива `options`.

Описание

Этот метод добавляет новый элемент `<option>` к данному элементу `<select>`. Аргумент *элемент* – это HTMLOptionElement, представляющий добавляемый элемент `<option>`. Аргумент *перед* задает HTMLOptionElement, перед которым должен быть добавлен *элемент*. Если *перед* является частью *OPTGROUP*, *элемент* всегда вставляется как часть той же группы. Если *перед* равен `null`, *элемент* становится последним дочерним узлом элемента `<select>`.

См. также

Объект Select в «Справочнике по клиентскому JavaScript»

HTMLSelectElement.blur() DOM Level 1 HTML

забирает клавиатурный фокус от этого элемента

Синтаксис

```
void blur();
```

Описание

Этот метод забирает клавиатурный фокус у данного элемента.

HTMLSelectElement.focus() DOM Level 1 HTML

передает клавиатурный фокус этому элементу

Синтаксис

```
void focus();
```

Описание

Этот метод передает клавиатурный фокус в данный элемент `<select>`, чтобы пользователь мог взаимодействовать с ним при помощи клавиатуры, а не мыши.

HTMLSelectElement.remove() DOM Level 1 HTML

удаляет опцию

Синтаксис

```
void remove(long индекс);
```

Аргументы

индекс

Позиция удаляемого элемента `<option>` внутри массива `options`.

Описание

Этот метод удаляет элемент `<option>` в указанной позиции массива `options`. Если указанный *индекс* меньше нуля или больше или равен количеству опций, метод `remove()` игнорирует его и не делает ничего.

См. также

Объект Select в «Справочнике по клиентскому JavaScript»

HTMLTableCaptionElement DOM Level 1 HTML

шапка таблицы HTML **Node → Element → HTMLElement → HTMLTableCaptionElement**

Свойства

`deprecated String align`

Горизонтальное выравнивание шапки относительно таблицы. Значение атрибута `align`. Признано устаревшим в пользу стилей CSS.

Описание

Элемент `<caption>` в таблице HTML.

См. также

Тип: HTMLTableElement.caption

HTMLTableCellElement

DOM Level 1 HTML

Ячейка таблицы HTML <td> или <th>

Node → Element → HTMLElement → HTMLTableCellElement

Свойства

`readonly long cellIndex`

Позиция этой клетки внутри строки.

В дополнение к свойству `cellIndex` этот интерфейс определяет свойства, приведенные в следующей таблице, которые непосредственно соответствуют атрибутам HTML элементов `<td>` и `<th>`.

Свойство	Атрибут	Описание
String abbr	abbr	См. спецификацию HTML
String align	align	Горизонтальное выравнивание ячейки
String axis	axis	См. спецификацию HTML
deprecated String bgColor	bgcolor	Цвет фона ячейки
String ch	char	Символ выравнивания
String chOff	choff	Смещение символа выравнивания
long colSpan	colspan	Столбцы, охваченные ячейкой
String headers	headers	Значения id для заголовков этой ячейки
deprecated String height	height	Высота ячейки в пикселах
deprecated boolean noWrap	nowrap	Ячейка, в которой не выполняется перенос слов
long rowSpan	rowspan	Строки, охваченные ячейкой
String scope	scope	Диапазон этой ячейки заголовка
String vAlign	valign	Вертикальное выравнивание ячейки
deprecated String width	width	Ширина ячейки в пикселах

Описание

Этот интерфейс представляет элементы `<td>` и `<th>` в таблицах HTML.

HTMLTableColElement

DOM Level 1 HTML

<col> или <colgroup> в таблице HTML

Node → Element → HTMLElement → HTMLTableColElement

Свойства

Этот интерфейс определяет свойства, перечисленные в следующей таблице, каждое из которых соответствует атрибуту HTML элемента `<col>` или `<colgroup>`.

Свойство	Атрибут	Описание
String align	align	Горизонтальное выравнивание по умолчанию
String ch	char	Символ выравнивания по умолчанию
String chOff	choff	Смещение выравнивания по умолчанию
long span	span	Количество столбцов, представляемых этим элементом
String vAlign	valign	Вертикальное выравнивание по умолчанию
String width	width	Ширина столбца (столбцов)

Описание

Этот интерфейс представляет элемент <col> или <colgroup> в таблице HTML.

HTMLTableElement

DOM Level 1 HTML

таблица в HTML-документе **Node → Element → HTMLElement → HTMLTableElement**

Свойства

`HTMLTableCaptionElement caption`

Ссылка на элемент <caption> таблицы или `null`, если он отсутствует.

`readonly HTMLCollection rows`

Массив (HTMLCollection) объектов HTMLTableRowElement, представляющих все строки в таблице. Он включает все строки, определенные с помощью тегов <thead>, <tfoot> и <tbody>.

`readonly HTMLCollection tBodies`

Массив (HTMLCollection) объектов HTMLTableSectionElement, представляющий все разделы <tbody> в этой таблице.

`HTMLTableSectionElement tFoot`

Элемент <tfoot> таблицы или `null`, если он отсутствует.

`HTMLTableSectionElement tHead`

Элемент <thead> таблицы или `null`, если он отсутствует.

В дополнение к только что перечисленным свойствам этот интерфейс определяет свойства, приведенные в следующей таблице и представляющие HTML-атрибуты элемента <table>.

Свойство	Атрибут	Описание
deprecated String align	align	Горизонтальное выравнивание таблицы в документе
deprecated String bgColor	bgcolor	Цвет фона таблицы
String border	border	Ширина границы вокруг таблицы
String cellPadding	cellpadding	Пространство между содержимым и границей ячейки
String cellSpacing	cellspacing	Пространство между границами ячеек
String frame	frame	Управляет отображением границ таблицы
String rules	rule	Управляет отображением линий внутри таблицы
String summary	summary	Общее описание таблицы
String width	width	Ширина таблицы

Методы

`createCaption()`

Возвращает элемент <caption> для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`createTFoot()`

Возвращает существующий элемент `<tfoot>` для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`createTHead()`

Возвращает существующий элемент `<thead>` для таблицы или создает (и вставляет) новый элемент, если он еще не существует.

`deleteCaption()`

Удаляет элемент `<caption>` из таблицы, если он существует.

`deleteRow()`

Удаляет строку в указанной позиции таблицы.

`deleteTFoot()`

Удаляет элемент `<tfoot>` из таблицы, если он существует.

`deleteTHead()`

Удаляет элемент `<thead>` из таблицы, если он существует.

`insertRow()`

Вставляет новый, пустой элемент `<tr>` в указанной позиции.

Описание

Этот интерфейс представляет элемент HTML `<table>` и определяет несколько удобных свойств и методов для получения и модификации различных секций таблицы. Эти методы и свойства облегчают работу с таблицами, но они также могут быть продублированы с помощью базовых методов DOM.

См. также

HTMLTableCaptionElement, HTMLTableCellElement, HTMLTableColElement, HTMLTableRowElement, HTMLTableSectionElement

HTMLTableElement.createCaption() DOM Level 1 HTML

получает или создает шапку таблицы

Синтаксис

```
HTMLElement createCaption();
```

Возвращаемое значение

Объект HTMLTableCaptionElement, представляющий элемент `<caption>` для этой таблицы. Если в таблице уже есть шапка, этот метод просто возвращает ее. Если в таблице еще нет шапки, этот метод создает новый (пустой) элемент `<caption>`, вставляет его в таблицу, а затем возвращает его.

HTMLTableElement.createTFoot() DOM Level 1 HTML

получает или создает <tfoot>

Синтаксис

```
HTMLElement createTFoot();
```

Возвращаемое значение

Объект HTMLTableSectionElement, представляющий элемент `<tfoot>` для этой таблицы. Если в таблице уже есть нижний колонтитул, этот метод просто возвращает ее. Если в таблице еще нет нижнего колонтитула, этот метод создает новый (пустой) элемент `<tfoot>`, вставляет его в таблицу, а затем возвращает его.

HTMLTableElement.createTHead() DOM Level 1 HTML

получает или создает <thead>

Синтаксис

```
HTMLElement createTHead();
```

Возвращаемое значение

Объект HTMLTableSectionElement, представляющий элемент `<thead>` для этой таблицы. Если в таблице уже есть верхний колонтитул, этот метод просто возвращает ее. Если в таблице еще нет верхнего колонтитула, этот метод создает новый (пустой) элемент `<thead>`, вставляет его в таблицу, а затем возвращает его.

HTMLTableElement.deleteCaption() DOM Level 1 HTML

удаляет шапку таблицы

Синтаксис

```
void deleteCaption();
```

Описание

Если в этой таблице имеется элемент `<caption>`, этот метод удаляет его из дерева документа. В противном случае он не делает ничего.

HTMLTableElement.deleteRow() DOM Level 1 HTML

удаляет строку таблицы

Синтаксис

```
void deleteRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Указывает позицию удаляемой строки внутри таблицы.

Исключения

Этот метод генерирует исключение DOMException с кодом, равным `INDEX_SIZE_ERR`, если *индекс* меньше нуля или больше или равен количеству строк в таблице.

Описание

Этот метод удаляет строку в указанной позиции в таблице. Строки нумеруются в том порядке, в котором они присутствуют в исходном тексте документа. Строки в секциях `<thead>` и `<tfoot>` нумеруются вместе со всеми другими строками в таблице.

См. также

HTMLTableSectionElement.deleteRow()

HTMLTableElement.deleteTFoot() DOM Level 1 HTML

удаляет из таблицы элемент <tfoot>

Синтаксис

```
void deleteTFoot();
```

Описание

Если в этой таблице есть элемент `<tfoot>`, этот метод удаляет его из дерева документа. Если таблица не имеет нижнего колонтитула, этот метод не делает ничего.

HTMLTableElement.deleteTHead() DOM Level 1 HTML

удаляет <thead> таблицы

Синтаксис

```
void deleteTHead();
```

Описание

Если в этой таблице есть элемент `<thead>`, этот метод удаляет его из дерева документа. Если таблица не имеет верхнего колонтитула, этот метод не делает ничего.

HTMLTableElement.insertRow() DOM Level 1 HTML

добавляет новую, пустую строку в таблицу

Синтаксис

```
HTMLElement insertRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция, в которую должна быть вставлена новая строка.

Возвращаемое значение

HTMLTableRowElement, представляющий только что вставленную строку.

Исключения

Этот метод генерирует исключение DOMException с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля или больше количества строк в таблице.

Описание

Этот метод создает новый элемент HTMLTableRowElement, представляющий тег `<tr>`, и вставляет его в таблицу в указанной позиции.

Новая строка вставляется в той же секции и непосредственно перед существующей строкой, в позиции, заданной аргументом *индекс*. Если *индекс* равен количеству строк в таблице, новая строка дописывается к последней секции таблицы. Если таблица первоначально пуста, новая строка вставляется в новую секцию `<tbody>`, которая, в свою очередь, вставляется в таблицу.

Для добавления содержимого к только что созданной строке можно использовать вспомогательный метод `HTMLTableRowElement.insertCell()`.

См. также

HTMLTableSectionElement.insertRow()

HTMLTableRowElement

DOM Level 1 HTML

элемент <tr> в таблице HTML **Node → Element → HTMLElement → HTMLTableRowElement**

Свойства

`readonly HTMLCollection cells`

Массив (HTMLCollection) объектов HTMLTableCellElement, представляющих ячейки в этой строке.

`readonly long rowIndex`

Позиция этой строки в таблице.

`readonly long sectionRowIndex`

Позиция этой строки в данном разделе (то есть внутри этого элемента `<thead>`, `<tbody>` или `<tfoot>`).

В дополнение к только что перечисленным свойствам этот интерфейс также определяет свойства, приведенные в следующей таблице, которые соответствуют HTML-атрибутам элемента `<tr>`.

Свойство	Атрибут	Описание
`String align`	`align`	Горизонтальное выравнивание ячеек, установленное по умолчанию в этой строке
`deprecated String bgColor`	`bgcolor`	Цвет фона этой строки, установленный по умолчанию
`String ch`	`char`	Символ выравнивания для ячеек этой строки
`String chOff`	`choff`	Смещение символа выравнивания для ячеек этой строки
`String vAlign`	`valign`	Вертикальное выравнивание ячеек, установленное по умолчанию в этой строке

Методы

`deleteCell()`

Удаляет указанную ячейку из этой строки.

`insertCell()`

Вставляет пустой элемент `<td>` в эту строку в указанной позиции.

Описание

Этот интерфейс представляет строку в таблице HTML.

HTMLTableRowElement.deleteCell()

DOM Level 1 HTML

удаляет ячейку из строки таблицы

Синтаксис

```
void deleteCell(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция удаляемой клетки в строке.

Исключения

Этот метод генерирует исключение DOMException с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля либо больше или равен количеству ячеек в строке.

Описание

Этот метод удаляет ячейки в указанной позиции в строке таблицы.

HTMLTableRowElement.insertCell()

DOM Level 1 HTML

вставляет новый, пустой элемент <td> в строку таблицы

Синтаксис

```
HTMLElement insertCell(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция, в которую должна быть вставлена новая ячейка.

Возвращаемое значение

Объект HTMLTableCellElement, представляющий только что созданный и вставленный элемент `<td>`.

Исключения

Этот метод генерирует исключение DOMException с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля или больше, чем количество ячеек в строке.

Описание

Этот метод создает новый элемент `<td>` и вставляет его в строку в указанной позиции. Новая ячейка вставляется непосредственно перед ячейкой, находящейся в данный момент в позиции, заданной аргументом *индекс*. Если *индекс* равен количеству ячеек в строке, новая ячейка дописывается в конец строки.

Обратите внимание, что этот вспомогательный метод позволяет вставлять только ячейки данных – `<td>`. Для того чтобы вставить ячейку верхнего колонтитула в строку, необходимо создать и вставить элемент `<th>` с помощью `Document.createElement()`, `Node.insertBefore()` и других связанных методов.

HTMLTableSectionElement

DOM Level 1 HTML

секция верхнего, нижнего колонтитула или тела таблицы

Node → Element → HTMLElement → HTMLTableSectionElement

Свойства

`readonly HTMLCollection rows`

Массив (HTMLCollection) объектов HTMLTableRowElement, представляющих строки в этой секции таблицы.

В дополнение к свойству rows этот интерфейс определяет свойства, приведенные в следующей таблице, которые представляют атрибуты данного элемента HTML.

Свойство	Атрибут	Описание
String align	align	Горизонтальное выравнивание ячеек, установленное по умолчанию в этой секции таблицы
String ch	char	Символ выравнивания, установленный по умолчанию для ячеек в этой секции
String chOff	choff	Смещение выравнивания, установленное по умолчанию для ячеек в этой секции
String vAlign	valign	Вертикальное выравнивание ячеек, установленное по умолчанию в этой секции таблицы

Методы

deleteRow()

Удаляет строку с указанным номером из этой секции.

insertRow()

Вставляет пустую строку в эту секцию в указанной позиции.

Описание

Этот интерфейс представляет секцию <tbody>, <thead> или <tfoot> таблицы HTML.

См. также

Тип: HTMLTableElement.tFoot, HTMLTableElement.tHead

HTMLTableSectionElement.deleteRow() DOM Level 1 HTML

удаляет строку внутри секции таблицы

Синтаксис

```
void deleteRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция в строке внутри этой секции.

Исключения

Этот метод генерирует исключение DOMException с кодом INDEX_SIZE_ERR, если *индекс* меньше нуля либо больше или равен количеству строк в этой секции.

Описание

Этот метод удаляет строку из указанной позиции в этой секции. Обратите внимание, что для этого метода аргумент *индекс* задает позицию строки внутри секции, а не в таблице в целом.

См. также

HTMLTableElement.deleteRow()

HTMLTableSectionElement.insertRow() DOM Level 1 HTML

вставляет новую, пустую строку в секцию таблицы

Синтаксис

```
HTMLElement insertRow(long индекс)
    throws DOMException;
```

Аргументы

индекс

Позиция внутри секции, в которую должна быть вставлена новая строка.

Возвращаемое значение

Элемент HTMLTableRowElement, представляющий только что созданный и вставленный элемент `<tr>`.

Исключения

Этот метод генерирует исключение DOMException с кодом `INDEX_SIZE_ERR`, если *индекс* меньше нуля или больше количества строк в этой секции.

Описание

Этот метод создает новый элемент `<tr>` и вставляет его в эту секцию таблицы в указанной позиции. Если аргумент *индекс* равен количеству строк в секции, новая строка дописывается в конец секции. В противном случае новая строка вставляется непосредственно перед строкой, находящейся в данный момент в позиции, заданной аргументом *индекс*. Обратите внимание, что для этого метода *индекс* задает позицию строки внутри одной секции таблицы, а не в таблице в целом.

См. также

HTMLTableElement.insertRow()

HTMLTextAreaElement DOM Level 1 HTML

элемент <textarea> в форме HTML **Node → Element → HTMLElement → HTMLTextAreaElement**

Свойства

`String accessKey`

Клавиша быстрого доступа (один символ), посредством которой веб-броузер может передать клавиатурный фокус в этот элемент. Соответствует атрибуту `accesskey`.

`long cols`

Ширина этого элемента в столбцах символов. Соответствует атрибуту `cols`.

`String defaultValue`

Первоначальное содержимое текстовой области. Когда форма сбрасывается, значение текстовой области восстанавливается к этому значению. Установка этого свойства приводит к изменению отображаемого в текстовой области текста.

`boolean disabled`

Если оно равно `true`, этот элемент отключается и пользователь не может с ним взаимодействовать. Соответствует атрибуту `disabled`.

`readonly HTMLFormElement form`

HTMLFormElement, представляющий элемент `<form>`, в котором содержится эта текстовая область, или `null`, если этот элемент не находится внутри формы.

`String name`

Имя этого элемента `<textarea>`, указанное в атрибуте `name`.

`boolean readOnly`

Если это свойство равно `true`, данный элемент доступен только для чтения и пользователь не может редактировать какой-либо отображаемый в нем текст. Соответствует атрибуту `readonly`.

`long rows`

Высота текстовой области в строках текста. Соответствует атрибуту `rows`.

`long tabIndex`

Позиция этого элемента в порядке обхода по Tab. Соответствует атрибуту `tabindex`.

`readonly String type`

Тип этого элемента, оставлен для совместимости с объектами HTMLInputElement. Значение этого свойства всегда равно «textarea».

`String value`

Текст, отображаемый в данный момент в текстовой области.

Методы

`blur()`

Забирает клавиатурный фокус от элемента.

`focus()`

Передает клавиатурный фокус этому элементу.

`select()`

Отмечает все содержимое текстовой области.

Описание

Этот интерфейс представляет элемент `<textarea>` для создания многострочного текстового поля в форме HTML. Начальное содержимое текстовой области задается между тегами `<textarea>` и `</textarea>`. Пользователь может редактировать это значение, а также получать и устанавливать текст с помощью свойства `value` (или путем изменения дочернего узла Text этого элемента).

См. также

HTMLFormElement, HTMLInputElement; Textarea в «Справочнике по клиентскому JavaScript»; главу 15

HTMLTextAreaElement.blur() DOM Level 1 HTML

забирает клавиатурный фокус от этого элемента

Синтаксис

```
void blur();
```

Описание

Этот метод забирает клавиатурный фокус от данного элемента.

HTMLTextAreaElement.focus() DOM Level 1 HTML

передает клавиатурный фокус этому элементу

Синтаксис

```
void focus();
```

Описание

Этот метод передает клавиатурный фокус данному элементу, чтобы пользователь мог редактировать отображаемый в нем текст без предварительного нажатия кнопки мыши на текстовой области.

HTMLTextAreaElement.select() DOM Level 1 HTML

выделяет текст в этом элементе

Синтаксис

```
void select();
```

Описание

Данный метод отмечает весь текст, отображаемый в этом элементе `<textarea>`. В большинстве броузеров это значит, что текст выделяется, и новый текст, введенный пользователем, заменит выделенный текст, а не будет дописан к нему.

LinkStyle DOM Level 2 StyleSheets

таблица стилей, связанная с узлом

Свойства

`readonly StyleSheet sheet`

Объект StyleSheet, связанный с этим узлом.

Описание

В реализациях DOM, поддерживающих модуль StyleSheets, этот интерфейс реализуется любым узлом, который ссылается на таблицу стилей, или определяет встроенную таблицу стилей. В этом случае свойство `sheet` предоставляет способ получения объекта StyleSheet, связанного с узлом.

В документах HTML этот интерфейс реализуют элементы `<style>` и `<link>`. Эти элементы представляются интерфейсами HTMLStyleElement и HTMLLinkElement, для которых нет отдельных статей в этом справочнике. Дополнительную информацию об этих интерфейсах можно найти в статье «HTMLElement».

В документы XML таблицы стилей включаются с помощью инструкции обработки. Подробности см. в статье «ProcessingInstruction».

См. также

HTMLElement, ProcessingInstruction

MediaList

DOM Level 2 StyleSheets

список типов носителей в таблице стилей

Свойства

`readonly unsigned long length`

Длина массива; количество типов носителей в списке.

`String mediaText`

Текстовое представление полного списка носителей (через запятую). Установка этого свойства может генерировать исключение DOMException с кодом `SYNTAX_ERR`, если новое значение содержит синтаксическую ошибку, или с кодом `NO_MODIFICATION_ALLOWED_EXCEPTION`, если список носителей доступен только для чтения.

Методы

`appendMedium()`

Добавляет новый тип носителя в конец списка.

`deleteMedium()`

Удаляет указанный тип носителя из списка.

`item()`

Возвращает тип носителя в указанной позиции в списке или `null`, если указано недопустимое значение индекса. В JavaScript объект MediaList также может рассматриваться как массив, и к нему можно обращаться с помощью обычной нотации с квадратными скобками, а не вызывать этот метод.

Описание

Этот интерфейс представляет список или массив типов носителей для таблицы стилей. `length` задает количество элементов в списке, а `item()` позволяет извлечь определенный тип носителя по его позиции. Методы `appendMedium()` и `deleteMedium()` позволяют дописывать и удалять типы носителей в списке. JavaScript позволяет рассматривать объект MediaList как массив, и вместо вызова метода `item()` можно использовать нотацию с квадратными скобками.

Стандарт HTML 4 определяет следующие типы носителей (они чувствительны к регистру и должны набираться строчными буквами): `screen`, `tty`, `tv`, `projection`, `handheld`, `print`, `braile`, `aural` и `all`. Тип `screen` более всего соответствует документам, отображаемым в веб-броузерах на настольных или переносных компьютерах. Тип `print` используется для стилей, предназначенных для печатных документов.

См. также

Тип: StyleSheet.media

MediaList.appendMedium()

DOM Level 2 StyleSheets

добавляет новый тип носителей в список

Синтаксис

```
void appendMedium(String новыйНоситель)
    throws DOMException;
```

Аргументы

новыйНоситель

Имя нового добавляемого типа носителя. Набор допустимых имен типов носителей см. на справочной странице «MediaList».

Исключения

Этот метод может генерировать исключение DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если список носителей доступен только для чтения, или `INVALID_CHARACTER_ERR`, если указанный аргумент *новыйНоситель* содержит недопустимые символы.

Описание

Этот метод дописывает указанный аргумент *новыйНоситель* в конец MediaList. Если MediaList уже содержит указанный тип носителя, он сначала удаляется из текущей позиции, а затем дописывается в конец списка.

MediaList.deleteMedium() DOM Level 2 StyleSheets

удаляет тип носителя из списка

Синтаксис

```
void deleteMedium(String старыйНоситель)
    throws DOMException;
```

Аргументы

старыйНоситель

Имя типа носителя, который должен быть удален из списка. Набор допустимых имен типов носителей см. на справочной странице «MediaList».

Исключения

Этот метод генерирует исключение DOMException с кодом `NOT_FOUND_ERR`, если в списке нет указанного типа *старыйНоситель*, или `NO_MODIFICATION_ALLOWED_ERR`, если список носителей доступен только для чтения.

Описание

Этот метод удаляет указанный тип носителя из данного объекта MediaList или генерирует исключение, если в списке нет указанного типа.

MediaList.item() DOM Level 2 StyleSheets

доступ по индексу к массиву типов носителей

Синтаксис

```
String item(unsigned long индекс);
```

Аргументы

индекс

Позиция требуемого типа носителя внутри массива.

Возвращаемое значение

Тип носителя (строка) в указанной позиции внутри MediaList или `null`, если *индекс* отрицателен или больше или равен `length`. Обратите внимание, что в JavaScript обычно

проще рассматривать объект MediaList как массив и обращаться к нему с помощью нотации массивов с использованием квадратных скобок, а не вызывать этот метод.

MouseEvent

DOM Level 2 Events

сведения о событии мыши **Event → UIEvent → MouseEvent**

Свойства

`readonly boolean altKey`

Была ли нажата клавиша Alt, когда произошло событие. Определено для всех типов событий мыши.

`readonly unsigned short button`

Какая кнопка мыши изменила состояние во время события mousedown, mouseup или click. Значение 0 обозначает левую кнопку, 2 – правую кнопку, а значение 1 – среднюю кнопку мыши. Обратите внимание, что это свойство определено, когда кнопка меняет состояние. Оно, например, не используется для сообщения о том, нажата ли кнопка во время события `mousemove`. Кроме того, это свойство не является битовой маской – оно ничего не сообщит, если нажата более чем одна кнопка.

В Netscape 6.0 и 6.01 вместо значений 0, 1 и 2 определены значения 1, 2 и 3. Эта ошибка исправлена в Netscape 6.1 .

`readonly long clientX, clientY`

Числа, задающие координаты X и Y курсора мыши относительно «клиентской области» или окна броузера. Обратите внимание, что эти координаты не учитывают прокрутку документа; если событие происходит на верхнем краю окна, `clientY` равно 0 независимо от того, как далеко выполнена прокрутка документа. Эти свойства определены для всех типов событий мыши.

`readonly boolean ctrlKey`

Была ли нажата клавиша Ctrl, когда произошло событие. Определено для всех типов событий мыши.

`readonly boolean metaKey`

Была ли нажата клавиша Meta, когда произошло событие. Определено для всех типов событий мыши.

`readonly EventTarget relatedTarget`

Ссылается на узел, который имеет отношение к целевому узлу события. Для событий mouseover это узел, из которого вышел курсор мыши при перемещении над целевым узлом. Для событий mouseout это узел, куда вошел курсор мыши, когда он вышел из целевого узла. Для других типов событий мыши `relatedTarget` не определено.

`readonly long screenX, screenY`

Числа, задающие координаты X и Y курсора мыши относительно верхнего левого угла монитора пользователя. Эти свойства определены для всех типов событий мыши.

`readonly boolean shiftKey`

Была ли нажата клавиша Shift, когда произошло событие. Определено для всех типов событий мыши.

Методы

`initMouseEvent()`

Инициализирует свойства только что созданного объекта MouseEvent.

Описание

Этот интерфейс определяет тип объекта Event, который передается событиям типа click, mousedown, mousemove, mouseout, mouseover и mouseup. Обратите внимание, что в дополнение к свойствам, перечисленным здесь, этот интерфейс также наследует свойства интерфейсов UIEvent и Event.

См. также

Event, UIEvent; главу 19

MouseEvent.initMouseEvent() DOM Level 2 Events

инициализирует свойства объекта MouseEvent

Синтаксис

```
void initMouseEvent(String typeArg,
                    boolean canBubbleArg,
                    boolean cancelableArg,
                    AbstractView viewArg,
                    long detailArg,
                    long screenXArg,
                    long screenYArg,
                    long clientXArg,
                    long clientYArg,
                    boolean ctrlKeyArg,
                    boolean altKeyArg,
                    boolean shiftKeyArg,
                    boolean metaKeyArg,
                    unsigned short buttonArg,
                    EventTarget relatedTargetArg);
```

Аргументы

Многочисленные аргументы этого метода задают начальные значения свойств данного объекта MouseEvent, включая свойства, унаследованные от интерфейсов Event и UIEvent. Имя каждого аргумента ясно указывает свойство, для которого указано значение, поэтому здесь они отдельно не перечисляются.

Описание

Этот метод инициализирует различные свойства только что созданного объекта MouseEvent. Он может вызываться только для объекта MouseEvent, созданного с помощью `Document.createEvent()`, и только до того, как этот объект будет передан методу `EventTarget.dispatchEvent()`.

MutationEvent

DOM Level 2 Events

сведения об изменении документа **Event → MutationEvent**

Константы

Возможные значения констант для свойства `attrChange`:

`unsigned short MODIFICATION = 1`

Узел Attr был модифицирован.

`unsigned short ADDITION = 2`

Узел Attr был добавлен.

`unsigned short REMOVAL = 3`

Узел Attr был удален.

Свойства

`readonly unsigned short attrChange`

Как был изменен атрибут (для событий DOMAttrModified). Три возможных значения определены в разделе «Константы».

`readonly String attrName`

Имя атрибута было изменено (для событий DOMAttrModified).

`readonly String newValue`

Новое значение узла Attr для событий DOMAttrModified или новое текстовое значение узла Text, Comment, CDATASection или ProcessingInstruction для событий DOMCharacterDataModified.

`readonly String prevValue`

Предыдущее значение узла Attr для событий DOMAttrModified или предыдущее значение узла Text, Comment, CDATASection или ProcessingInstruction для событий DOMCharacterDataModified.

`readonly Node relatedNode`

Узел Attr для событий DOMAttrModified или родительский узел вставленного или удаленного узла для событий DOMNodeInserted и DOMNodeRemoved.

Методы

`initMutationEvent()`

Инициализирует свойства только что созданного объекта MutationEvent.

Описание

Этот интерфейс определяет тип объекта Event, который передается событиями перечисленных здесь типов (обратите внимание, что ни один из этих типов событий не может быть отменен с помощью `Event.preventDefault()`):

DOMAttrModified

Генерируется, когда добавляется, удаляется или изменяется атрибут элемента документа. Целью события является элемент, содержащий атрибут, и всплывание события происходит от него.

DOMCharacterDataModified

Генерируется, когда изменяются символьные данные в узле Text, Comment, CDATASection или ProcessingInstruction. Целью события является измененный узел, и всплывание события происходит от него.

DOMNodeInserted

Генерируется после того, как узел добавляется в качестве дочернего для другого узла. Целью события является узел, который был вставлен, и отсюда событие всплывает по дереву документа. Свойство `relatedNode` задает новый родительский узел вставленного узла. Это событие не генерируется для каких-либо потомков вставленного узла.

DOMNodeInsertedIntoDocument

Генерируется после того, как узел вставляется в дерево документа, как для узлов, непосредственно вставленных в дерево, так и для узлов, вставленных косвенным образом при вставке их предка. Целью этого события является вставленный узел. Так как события этого типа могут быть нацелены на любой узел в поддереве, они не всплывают.

DOMNodeRemoved

Генерируется непосредственно перед тем, как узел удаляется из родительского узла. Целью события является удаляемый узел, и отсюда событие всплывает вверх по дереву документа. Свойство `relatedNode` содержит родительский узел, из которого был удален данный узел.

DOMNodeRemovedFromDocument

Генерируется непосредственно перед тем, как узел удаляется из дерева документа. Для непосредственно удаленного узла и каждого из его потомков генерируются отдельные события. Целью события является узел, который должен быть удален. События этого типа не всплывают.

DOMSubtreeModified

Генерируются как итоговое событие, когда вызов метода DOM приводит к возбуждению нескольких мутационных событий. Целью этого события является глубже всего вложенный общий предок всех изменений, произошедших в документе, и от этой точки оно всплывает вверх по дереву документа. Если вам не интересны детали изменений, а требуется лишь уведомление о том, какие части документа были изменены, можете предпочесть регистрировать слушатели для этого типа событий.

MutationEvent.initMutationEvent() DOM Level 2 Events

инициализирует свойства нового MutationEvent

Синтаксис

```
void initMutationEvent(String typeArg,
                       boolean canBubbleArg,
                       boolean cancelableArg,
                       Node relatedNodeArg,
                       String prevValueArg,
                       String newValueArg,
                       String attrNameArg,
                       unsigned short attrChangeArg);
```

Аргументы

Различные аргументы этого метода задают начальные значения свойств этого объекта MutationEvent, включая свойства, унаследованные от интерфейса Event. Имя каждого аргумента ясно указывает свойство, для которого указано значение, поэтому здесь они отдельно не перечисляются.

Описание

Этот метод инициализирует различные свойства только что созданного объекта MutationEvent. Он может вызываться только для объекта MutationEvent, созданного с помощью `Document.createEvent()`, и только до того, как этот объект будет передан методу `EventTarget.dispatchEvent()`.

NamedNodeMap

DOM Level 1 Core

коллекция узлов, к которым можно обращаться по имени и по номеру позиции

Свойства

`readonly unsigned long length`

Количество узлов в массиве.

Методы

`getNamedItem()`

Ищет узел по имени.

`getNamedItemNS()` *[DOM Level 2]*

Ищет узел, заданный пространством имен и именем.

`item()`

Извлекает узел в указанной позиции внутри NamedNodeMap. В JavaScript это также можно сделать, используя позицию узла в качестве индекса массива.

`removeNamedItem()`

Удаляет узел с указанным именем из NamedNodeMap.

`removeNamedItemNS()` *[DOM Level 2]*

Удаляет узел, заданный именем и пространством имен, из NamedNodeMap.

`setNamedItem()`

Добавляет новый узел (или заменяет существующий) в NamedNodeMap. Свойство `nodeName` объекта Node выступает в качестве имени узла.

`setNamedItemNS()` *[DOM Level 2]*

Добавляет новый узел (или заменяет существующий) в NamedNodeMap. Свойства `namespaceURI` и `localName` объекта Node выступают в качестве имени узла.

Описание

Интерфейс NamedNodeMap определяет коллекцию узлов, которые можно искать по значению их свойства `nodeName` либо по свойству `namespaceURI` или `localName`, если узлы используют пространства имен.

Наиболее интересное применение интерфейса – это свойство `attributes` интерфейса Node: коллекция узлов Attr, в которой можно выполнять поиск по имени атрибута. Многие методы NamedNodeMap аналогичны методам Element для манипуляции ат-

рибутами. Атрибутами элементов обычно проще всего манипулировать через методы интерфейса Element, и интерфейс NamedNodeMap используется нечасто.

Объекты NamedNodeMap – «живые», то есть они немедленно реагируют на любые изменения в дереве документа. Например, если вы получите коллекцию NamedNodeMap, представляющую атрибуты элемента, и затем добавите новый атрибут к этому элементу, новый атрибут будет доступен через NamedNodeMap.

См. также

NodeList

Тип: DocumentType.entities, DocumentType.notations, Node.attributes

NamedNodeMap.getNamedItem() DOM Level 1 Core

поиск узла по имени

Синтаксис

```
Node getNamedItem(String имя);
```

Аргументы

имя

Значение свойства `nodeName` искомого узла.

Возвращаемое значение

Узел с указанным именем или `null`, если узел с таким именем не был найден.

NamedNodeMap.getNamedItemNS() DOM Level 2 Core

ищет узел по имени и пространству имен

Синтаксис

```
Node getNamedItemNS(String URIпространствоИмен,
                    String локальноеИмя);
```

Аргументы

URIпространствоИмен

Свойство `namespaceURI` искомого узла или `null` в случае отсутствия пространства имен.

локальноеИмя

Свойство `localName` локального узла.

Возвращаемое значение

Элемент коллекции NamedNodeMap, имеющий указанные значения свойств `namespaceURI` и `localName`, или `null`, если такого узла нет.

Описание

Метод `getNamedItemNS()` ищет элемент коллекции NamedNodeMap по пространству имен и локальному имени. Применение имеет смысл только для документов XML, использующих пространства имен.

NamedNodeMap.item() DOM Level 1 Core

возвращает элемент NamedNodeMap по его позиции

Синтаксис

```
Node item(unsigned long индекс);
```

Аргументы

индекс

Позиция или индекс нужного узла.

Возвращаемое значение

Узел в указанной позиции, или `null`, если *индекс* больше нуля или больше или равен длине NamedNodeMap.

Описание

Этот метод возвращает элемент с указанным именем из NamedNodeMap. В JavaScript объекты NamedNodeMap ведут себя как доступные только для чтения массивы, и можно не вызывать этот метод, а использовать позицию узла как индекс массива в квадратных скобках.

Несмотря на то что интерфейс NamedNodeMap позволяет выполнять цикл по его узлам, он не представляет собой упорядоченную коллекцию узлов. Любые изменения, внесенные в NamedNodeMap (такие как `removeNamedItem()` или `setNamedItem()`), могут привести к полному переупорядочиванию элементов. Следовательно, не нужно модифицировать NamedNodeMap во время обхода его элементов.

См. также

NodeList

NamedNodeMap.removeNamedItem() DOM Level 1 Core

удаляет узел, заданный именем

Синтаксис

```
Node removeNamedItem(String имя)
    throws DOMException;
```

Аргументы

имя

Свойство `nodeName` узла, который должен быть удален.

Возвращаемое значение

Узел, который был удален.

Исключения

Этот метод генерирует исключение DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если NamedNodeMap доступен только для чтения и не допускает удалений, или со свойством `code`, равным `NOT_FOUND_ERR`, если в NamedNodeMap не существует узла с указанным *именем*.

Описание

Удаляет узел с указанным именем из NamedNodeMap. Обратите внимание, что если NamedNodeMap представляет набор атрибутов для элемента, удаление узла Attr для атрибута, явно установленного в документе, может привести к тому, что удаленный узел Attr будет заменен новым Attr, представляющим значение по умолчанию (если оно существует) этого атрибута.

См. также

Element.removeAttribute()

NamedNodeMap.removeNamedItemNS() DOM Level 2 Core

удаляет узел, заданный пространством имен и именем

Синтаксис

```
Node removeNamedItemNS(String URIпространстваИмен,
                       String локальноеИмя)
    throws DOMException;
```

Аргументы

URIпространстваИмен

Свойство `namespaceURI` узла, который должен быть удален, или `null` в случае отсутствия пространства имен.

локальноеИмя

Свойство `localName` узла, который должен быть удален.

Возвращаемое значение

Узел, который был удален.

Исключения

Этот метод генерирует исключения по тем же причинам, что и `removeNamedItem()`.

Описание

Этот метод работает точно так же, как `removeNamedItem()`, за исключением того, что узел, который должен быть удален, задается пространством имен и локальным именем, а не просто именем. Его применение имеет смысл только для XML-документов, использующих пространства имен.

NamedNodeMap.setNamedItem() DOM Level 1 Core

добавляет или заменяет узел в NamedNodeMap

Синтаксис

```
Node setNamedItem(Node arg)
    throws DOMException;
```

Аргументы

arg

Узел, который должен быть добавлен в NamedNodeMap.

Возвращаемое значение

Узел, который был заменен, или `null`, если узел не был заменен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code`:

`HIERARCHY_REQUEST_ERR`

arg – это узел, тип которого не подходит для данной коллекции NamedNodeMap (например, это не узел Attr).

`INUSE_ATTRIBUTE_ERR`

arg – это узел Attr, уже связанный с элементом.

`NO_MODIFICATION_ALLOWED_ERR`

NamedNodeMap доступен только для чтения.

`WRONG_DOCUMENT_ERR`

Значение `ownerDocument` в *arg* отлично от документа, из которого был создан NamedNodeMap.

Описание

`setNamedItem()` добавляет указанный узел к NamedNodeMap и позволяет искать его по значению свойства `nodeName` узла. Если NamedNodeMap уже содержит узел с этим именем, данный узел заменяется и становится возвращаемым значением метода.

См. также

Element.setAttribute()

NamedNodeMap.setNamedItemNS() DOM Level 2 Core

добавляет узел в NamedNodeMap, использующий пространства имен

Синтаксис

```
Node setNamedItemNS(Node arg)
    throws DOMException;
```

Аргументы

arg

Узел, который должен быть добавлен в NamedNodeMap.

Возвращаемое значение

Узел, который был заменен, или `null`, если узел не был заменен.

Исключения

Узел генерирует исключения по тем же причинам, что и `setNamedItem()`. Он также может генерировать исключение DOMException с кодом `NOT_SUPPORTED_ERR`, если метод вызывается в реализации, не поддерживающей XML-документы или пространства имен XML.

Описание

Этот метод работает так же, как `setNamedItem()`, за исключением того, что узел, добавляемый в NamedNodeMap, можно затем искать по значениям его свойств `namespaceURI` и `localName`, а не по свойству `nodeName`. Этот метод применяется только при работе с документами XML, использующими пространства имен. Обратите внимание, что этот метод может не поддерживаться (то есть генерировать исключение) в реализациях, не поддерживающих XML-документы.

Node

DOM Level 1 Core

узел в дереве документа

Подынтерфейсы

Attr, CharacterData, Document, DocumentFragment, DocumentType, Element, Entity, EntityReference, Notation, ProcessingInstruction

Также реализует

EventTarget

Если реализация DOM поддерживает модуль Events, любой узел в дереве документа также реализует интерфейс EventTarget, и в нем могут быть зарегистрированы слушатели событий. Методы, определенные интерфейсом EventTarget, здесь не приводятся; подробности см. на справочных страницах «EventTarget» и «EventListener».

Константы

Все объекты Node реализуют один из перечисленных выше подынтерфейсов. В любом объекте Node имеется свойство `nodeType`, задающее, какой из подынтерфейсов этот объект реализует. Эти константы представляют собой допустимые значения данного свойства; их имена понятны без объяснений. Обратите внимание, что это статические свойства функции-конструктора `Node()`; они не являются свойствами индивидуальных объектов Node. Также обратите внимание, что они не поддерживаются в Internet Explorer 4, 5 и 6.

```
Node.ELEMENT_NODE = 1;                  // Element
Node.ATTRIBUTE_NODE = 2;                // Attr
Node.TEXT_NODE = 3;                     // Text
Node.CDATA_SECTION_NODE = 4;            // CDATASection
Node.ENTITY_REFERENCE_NODE = 5;         // EntityReference
Node.ENTITY_NODE = 6;                   // Entity
Node.PROCESSING_INSTRUCTION_NODE = 7;   // ProcessingInstruction
Node.COMMENT_NODE = 8;                  // Comment
Node.DOCUMENT_NODE = 9;                 // Document
Node.DOCUMENT_TYPE_NODE = 10;           // DocumentType
Node.DOCUMENT_FRAGMENT_NODE = 11;       // DocumentFragment
Node.NOTATION_NODE = 12;                // Notation
```

Свойства

`readonly NamedNodeMap attributes`

Если данный узел является элементом, это свойство задает атрибуты этого элемента. `attributes` — это объект NamedNodeMap, позволяющий получать атрибуты по имени или по номеру и возвращающий их в форме объектов Attr. На практике почти всегда проще использовать метод `getAttribute()` интерфейса Element для получения значения атрибута в виде строки. Обратите внимание, что возвращаемый объект NamedNodeMap «живой»: любые изменения в атрибутах этого элемента немедленно становятся видимыми через него.

`readonly Node[] childNodes`

Содержит дочерние узлы текущего узла. Это свойство никогда не должно быть равно `null`: для узлов, не имеющих дочерних узлов, `childNodes` — это массив со свойством `length`, равным нулю. Это свойство формально представляет собой объ-

ект NodeList, но ведет себя, как массив объектов Node. Обратите внимание, что возвращаемый объект NodeList «живой»: любое изменение в списке дочерних узлов элемента немедленно становится видимым через NodeList.

`readonly Node firstChild`

Первый дочерний узел этого узла или `null`, если узел не имеет дочерних узлов.

`readonly Node lastChild`

Последний дочерний узел данного узла или `null`, если у него нет дочерних узлов.

`readonly String localName` *[DOM Level 2]*

В документах XML, использующих пространства имен, задает локальную часть имени элемента или атрибута. Это свойство никогда не применяется при работе с документами HTML. См. также свойства `namespaceURI` и `prefix`.

`readonly String namespaceURI` *[DOM Level 2]*

В документах XML, использующих пространства имен, задает URI пространства имен узла Element или Attribute. Это свойство никогда не применяется при работе с документами HTML. См. также свойства `localName` и `prefix`.

`readonly Node nextSibling`

Смежный узел, непосредственно следующий за данным узлом в массиве `childNodes[]` узла `parentNode`, или `null`, если такого узла нет.

`readonly String nodeName`

Имя узла. Для узлов Element задает имя тега элемента, которое может быть также получено с помощью свойства `tagName` интерфейса Element. Для других типов узлов значение зависит от типа. Подробности см. в таблице, приведенной ниже, в разделе «Описание».

`readonly unsigned short nodeType`

Тип узла, то есть информация о том, какой подынтерфейс реализует узел. Допустимые значения определяются перечисленными выше константами. Однако так как эти константы не поддерживаются в Internet Explorer, вместо них могут применяться жестко закодированные значения. В документах HTML распространенные значения для этого свойства таковы: 1 – для узлов Element, 3 – для узлов Text, 8 – для узлов Comment и 9 – для единственного узла Document верхнего уровня.

`String nodeValue`

Значение узла. Для узлов Text содержит текстовое содержимое. Для других типов узлов значение зависит от `nodeType`, как показано в приведенной ниже таблице.

`readonly Document ownerDocument`

Объект Document, частью которого является узел. Для узлов Document это свойство равно `null`.

`readonly Node parentNode`

Родительский узел (или узел-контейнер) этого узла или `null`, если родительского узла не существует. Обратите внимание, что узлы Document и Attr никогда не имеют родительских узлов. Кроме того, в узлах, которые были удалены из документа, или в только что созданных узлах, которые еще не вставлены в дерево документа, `parentNode` равно `null`.

`String prefix` *[DOM Level 2]*

Для документов XML, использующих пространства имен, задает префикс пространства имен узла Element или Attribute. Это свойство никогда не применяется при работе с HTML-документами. См. также свойства `localName` и `namespaceURL`. Установка этого свойства может привести к исключению, если новое значение содержит недопустимые символы, имеет неверный формат или не соответствует свойству `namespaceURI`.

`readonly Node previousSibling`

Смежный узел, непосредственно предшествующий данному узлу в массиве `childNodes[]` родительского узла `parentNode`, или `null`, если такого узла нет.

Методы

`appendChild()`

Добавляет узел в дерево документа, дописывая его к массиву `childNodes[]` этого узла. Если узел уже находится в дереве документа, он удаляется и вставляется в новую позицию.

`cloneNode()`

Создает копию данного узла или узла и его потомков.

`hasAttributes()` *[DOM Level 2]*

Возвращает `true`, если узел является узлом Element и имеет какие-либо атрибуты.

`hasChildNodes()`

Возвращает `true`, если в этом узле имеются какие-либо дочерние элементы.

`insertBefore()`

Вставляет узел в дерево документа непосредственно перед указанным дочерним узлом данного узла. Если вставляемый узел уже имеется в дереве, он удаляется и вставляется в новую позицию.

`isSupported()` *[DOM Level 2]*

Возвращает `true`, если указанный номер версии указанного модуля поддерживается данным узлом.

`normalize()`

«Нормализует» все текстовые узлы-потомки этого узла путем удаления пустых узлов Text и объединения смежных узлов Text.

`removeChild()`

Удаляет (и возвращает) указанный дочерний узел из дерева документа.

`replaceChild()`

Удаляет (и возвращает) указанный дочерний узел из дерева документа, заменяя его другим узлом.

Описание

Все объекты в дереве документа (включая сам объект Document) реализуют интерфейс Node, который предоставляет фундаментальные свойства и узлы для обхода и манипуляции деревом. Узел `parentNode` и массив `childNodes[]` позволяют передвигаться вверх и вниз по дереву документа. Можно перечислить дочерние узлы данного узла, выполнив цикл по элементам `childNodes[]` или используя свойства `firstChild` и `nextSibling` (или свойства `lastChild` и `previousSibling` – для обхода в обратном порядке).

Методы appendChild(), insertBefore(), removeChild() и replaceChild() позволяют модифицировать дерево документа, изменяя дочерние узлы данного узла.

Каждый объект в дереве документа реализует как интерфейс Node, так и более специализированный интерфейс, такой как Element или Text. Свойство nodeType указывает, какой подынтерфейс реализует узел. Это свойство позволяет проверить тип узла перед тем, как использовать свойства и методы более специализированного интерфейса. Например:

```
var n;                        // Содержит узел, с которым мы работаем
if (n.nodeType == 1) {        // Сравниваем его тип с константой Node.ELEMENT_NODE
    var tagname = n.tagName;  // Если узел является узлом Element, это имя тега
}
```

Свойства nodeName и nodeValue задают дополнительную информацию об узле, но, как показано в следующей таблице, их значения зависят от nodeType. Обратите внимание, что подынтерфейсы обычно определяют специальные свойства (такие как свойство tagName узлов Element или свойство data узлов Text) для получения этой информации.

nodeType	nodeName	nodeValue
ELEMENT_NODE	Имя тега элемента	null
ATTRIBUTE_NODE	Имя атрибута	Значение атрибута
TEXT_NODE	#text	Текст узла
CDATA_SECTION_NODE	#cdata-section	Текст узла
ENTITY_REFERENCE_NODE	Имя сущности, на которую указывает ссылка	null
ENTITY_NODE	Имя сущности	null
PROCESSING_INSTRUCTION_ NODE	Цель инструкции обработки	Остальная часть инструкции обработки
COMMENT_NODE	#comment	Текст комментария
DOCUMENT_NODE	#document	null
DOCUMENT_TYPE_NODE	Имя типа документа	null
DOCUMENT_FRAGMENT_NODE	#document-fragment	null
NOTATION_NODE	Имя нотации	null

См. также

Document, Element, Text; главу 17

Node.appendChild()

DOM Level 1 Core

вставляет узел в качестве последнего дочернего узла данного узла

Синтаксис

```
Node appendChild(Node новыйДочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел, который должен быть вставлен в документ. Если это узел DocumentFragment, он не вставляется непосредственно, а вставляются его дочерние узлы.

Возвращаемое значение

Узел, который был добавлен.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`HIERARCHY_REQUEST_ERR`

Узел не допускает наличия дочерних узлов, или не допускает дочерних улов указанного типа, или *новыйДочерниийУзел* является предком данного узла (или им самим).

`WRONG_DOCUMENT_ERR`

Свойство `ownerDocument` *новогоДочернегоУзла* не совпадает со свойством `ownerDocument` данного узла.

`NO_MODIFICATION_ALLOWED_ERR`

Этот узел доступен только для чтения и не допускает добавления дочерних узлов, или добавляемый узел уже является частью дерева документа, а его родительский узел доступен только для чтения и не допускает удаления дочерних узлов.

Описание

Этот метод добавляет узел *новыйДочерниийУзел* в документ, вставляя его в качестве последнего дочернего узла. Если *новыйДочерниийУзел* уже присутствует в дереве документа, он удаляется из дерева и вставляется в новое место. Если *новыйДочерниийУзел* является узлом DocumentFragment, сам узел не вставляется, а вместо этого в конец массива `childNodes[]` данного узла вставляются по порядку все дочерние узлы DocumentFragment. Обратите внимание, что узел из другого документа (или созданный другим документом) не может быть вставлен в текущий документ. То есть свойство `ownerDocument` *новогоДочернегоУзла* должно совпадать со свойством `ownerDocument` данного узла.

Пример

Следующая функция вставляет новый абзац в конец документа:

```
function appendMessage(message) {
    var pElement = document.createElement("P");
    var messageNode = document.createTextNode(message);
    pElement.appendChild(messageNode);     // Добавляет текст к абзацу
    document.body.appendChild(pElement);   // Добавляет абзац к телу документа
}
```

См. также

Node.insertBefore(), Node.removeChild(), Node.replaceChild()

Node.cloneNode() DOM Level 1 Core

дублирует узел и (необязательно) всех его потомков

Синтаксис

`Node cloneNode(boolean` *глубина*`);`

Аргументы

глубина

Если этот аргумент равен `true`, `cloneNode()` рекурсивно «клонирует» всех потомков этого узла. В противном случае копируется только данный узел.

Возвращаемое значение

Копия этого узла.

Описание

Метод `cloneNode()` создает копию узла, для которого он вызван, и возвращает ее. Если ему передается аргумент `true`, он также рекурсивно копирует всех потомков этого узла. В противном случае он копирует только данный узел, но не его дочерние узлы. Возвращаемый узел не является частью дерева документа, а его свойство `parentNode` равно `null`. Когда копируется узел Element, то копируются и все его атрибуты. Заметьте, однако, что функции EventListener, зарегистрированные для узла, не копируются.

Node.hasAttributes() DOM Level 2 Core

Определяет, есть ли в узле атрибуты

Синтаксис

```
boolean hasAttributes();
```

Возвращаемое значение

`true`, если в узле имеется один или несколько атрибутов, и `false`, если их нет. Обратите внимание, что только узлы Element имеют атрибуты.

См. также

Element.getAttribute(), Element.hasAttribute(), Node.attributes

Node.hasChildNodes() DOM Level 1 Core

определяет, имеются ли в узле дочерние узлы

Синтаксис

```
boolean hasChildNodes();
```

Возвращаемое значение

`true`, если в этом узле есть один или более дочерних узлов, и `false`, если их нет.

См. также

Node.childNodes

Node.insertBefore() DOM Level 1 Core

вставляет узел в дерево документа перед указанным узлом

Синтаксис

```
Node insertBefore(Node новыйДочернийУзел,
                  Node дочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел, который должен быть вставлен в дерево. Если это узел DocumentFragment, вместо него вставляются его дочерние узлы.

дочерниийУзел

Дочерний узел данного узла, перед которым должен быть вставлен *новыйДочерниийУзел*. Если этот аргумент равен null, *новыйДочерниийУзел* вставляется в качестве последнего дочернего узла данного.

Возвращаемое значение

Узел, который был вставлен.

Исключения

Этот метод может генерировать исключение DOMException со следующими значениями code:

HIERARCHY_REQUEST_ERR

Этот узел не поддерживает дочерние узлы, или не допускает наличия дочерних узлов указанного типа, или *новыйДочерниийУзел* является предком данного узла (или самим узлом).

WRONG_DOCUMENT_ERR

Свойство ownerDocument узла *новыйДочерниийУзел* и данного узла не совпадают.

NO_MODIFICATION_ALLOWED_ERR

Этот узел доступен только для чтения и не допускает вставки, или родительский узел *новогоДочернегоУзла* доступен только для чтения и не допускает удаления.

NOT_FOUND_ERR

дочерниийУзел не является дочерним для данного узла.

Описание

Этот метод вставляет *новыйДочерниийУзел* в дерево документа в качестве дочернего узла данного. Новый узел позиционируется в массиве childNodes[] так, что оказывается непосредственно перед *дочернимУзлом*. Если *дочерниийУзел* равен null, *новыйДочерниийУзел* вставляется в конец childNodes[] точно так же, как в случае применения метода appendChild(). Обратите внимание, что не допускается вызов этого метода с аргументом *дочерниийУзел*, не являющимся дочерним узлом данного.

Если *новыйДочерниийУзел* уже присутствует в дереве документа, он удаляется из дерева и затем вставляется заново в новую позицию. Если *новыйДочерниийУзел* является узлом DocumentFragment, он не вставляется сам; вместо этого в указанную позицию вставляются по порядку все его дочерние узлы.

Пример

Следующая функция вставляет новый абзац в начало документа:

```
function insertMessage(message) {
    var paragraph = document.createElement("p");  // Создаем элемент <p>
    var text = document.createTextNode(message);  // Создаем узел Text
    paragraph.appendChild(text);                  // Добавляем текст в абзац
    // Теперь вставляем абзац перед первым дочерним узлом тела документа
    document.body.insertBefore(paragraph, document.body.firstChild)
}
```

См. также

Node.appendChild(), Node.removeChild(), Node.replaceChild()

Node.isSupported() DOM Level 2 Core

определяет, поддерживает ли узел данный модуль

Синтаксис

```
boolean isSupported(String модуль,
                    String версия);
```

Аргументы

модуль

Имя проверяемого модуля.

версия

Номер версии проверяемого модуля или пустая строка (проверка поддержки любой версии модуля).

Возвращаемое значение

`true`, если узел поддерживает указанную версию указанного модуля, или `false`, если не поддерживает.

Описание

Стандарт W3C DOM имеет модульную структуру, и реализации не обязаны поддерживать все модули или возможности стандарта. Этот метод проверяет, поддерживает ли реализация этого узла указанную версию модуля с указанным именем. Список значений аргументов *модуль* и *версия* приведен на справочной странице «DOMImplementation.hasFeature()».

См. также

DOMImplementation.hasFeature()

Node.normalize() DOM Level 1 Core

объединяет смежные узлы Text и удаляет пустые

Синтаксис

```
void normalize();
```

Описание

Данный метод обходит всех потомков этого узла и «нормализует» документ, удаляя любые пустые узлы Text и собирая все смежные узлы Text в один узел. Это иногда позволяет упростить структуру дерева после вставки и удаления узлов.

См. также

Text

Node.removeChild() DOM Level 1 Core

удаляет (и возвращает) указанный дочерний узел данного узла

Синтаксис

```
Node removeChild(Node дочернийУзел)
    throws DOMException;
```

Аргументы

дочернийУзел

Удаляемый дочерний узел.

Возвращаемое значение

Узел, который был удален.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`NO_MODIFICATION_ALLOWED_ERR`

Этот узел доступен только для чтения и не допускает удаления дочерних узлов.

`NOT_FOUND_ERR`

дочернийУзел не является дочерним для данного узла.

Описание

Этот метод удаляет указанный дочерний узел из массива `childNodes[]` данного узла. Вызов этого метода с узлом, не являющимся дочерним, будет ошибкой. Метод `removeChild()` возвращает *дочернийУзел* после его удаления. *дочернийУзел* продолжает быть действительным узлом и может быть позднее вставлен в документ.

Пример

Последний дочерний узел можно удалить из тела документа так:

```
document.body.removeChild(document.body.lastChild);
```

См. также

Node.appendChild(), Node.insertBefore(), Node.replaceChild()

Node.replaceChild() DOM Level 1 Core

заменяет дочерний узел новым узлом

Синтаксис

```
Node replaceChild(Node новыйДочернийУзел,
                  Node старыйДочернийУзел)
    throws DOMException;
```

Аргументы

новыйДочернийУзел

Узел для замены.

старыйДочернийУзел

Заменяемый узел.

Возвращаемое значение

Узел, который был удален из документа и заменен.

Исключения

Этот метод может генерировать исключение DOMException со следующими значениями `code`:

HIERARCHY_REQUEST_ERR

Узел не допускает наличия дочерних узлов, или не допускает дочерних узлов указанного типа, или *новыйДочерниймУзел* является предком данного узла (или им самим).

WRONG_DOCUMENT_ERR

Свойство ownerDocument *новогоДочернегоУзла* не совпадает со свойством ownerDocument данного узла.

NO_MODIFICATION_ALLOWED_ERR

Данный узел доступен только для чтения и не допускает замены, или *новыйДочерниймУзел* является дочерним для данного узла и не допускает удаления.

NOT_FOUND_ERR

старыйДочерниймУзел не является дочерним для данного узла.

Описание

Этот метод заменяет один узел в дереве документа на другой. *старыйДочерниймУзел* — это узел, который должен быть заменен, и он должен быть дочерним для данного узла. *новыйДочерниймУзел* — это узел, который занимает его место в массиве childNodes[] данного узла.

Если *новыйДочерниймУзел* уже является частью документа, то он сначала удаляется из документа перед повторной вставкой в новую позицию. Если *новыйДочерниймУзел* — это DocumentFragment, сам узел не вставляется, а вместо этого в позицию, ранее занятую *старымДочернимУзлом*, по порядку вставляются все его дочерние узлы DocumentFragment.

Пример

Следующий код заменяет узел n элементом <b>, а затем вставляет заменяемый узел в элемент <b>, в результате чего этот узел отображается жирным шрифтом:

```
// Получаем первый дочерний узел первого абзаца документа
var n = document.getElementsByTagName("p")[0].firstChild;
var b = document.createElement("b");  // Создает элемент <b>
n.parentNode.replaceChild(b, n);      // Заменяет узел узлом <b>
b.appendChild(n);                     // Вставляет узел как дочерний по  отношению к <b>
```

См. также

Node.appendChild(), Node.insertBefore(), Node.removeChild()

NodeFilter

DOM Level 2 Traversal

функция для фильтрации узлов дерева документа

Константы

Следующие три константы представляют собой допустимые возвращаемые значения для функций фильтрации узлов. Обратите внимание, что это статические свойства объекта с именем NodeFilter, а не свойства отдельных функций фильтрации узлов:

short FILTER_ACCEPT = 1

Принимать этот узел. NodeIterator или TreeWalker будут возвращать этот узел при выполнении обхода документа.

`short FILTER_REJECT = 2`

Отвергать этот узел. NodeIterator или TreeWalker будут вести себя так, как если бы этот узел не существовал. Кроме того, это значение информирует TreeWalker о необходимости игнорировать все дочерние узлы данного узла.

`short FILTER_SKIP = 3`

Пропустить данный узел. NodeIterator или TreeWalker не будут возвращать этот узел, но будут рекурсивно рассматривать его дочерние узлы при выполнении обхода документа.

Следующие константы представляют собой битовые флаги, которые могут быть установлены в аргументе *чтоПоказывать* методов `createNodeIterator()` и `createTreeWalker()` объекта Document. Каждая константа соответствует одному типу узлов Document (список типов узлов см. на справочной странице «Node») и указывает, что NodeIterator или TreeWalker должен рассматривать узлы этого типа при обходе документа. Несколько констант могут быть объединены с помощью оператора логического ИЛИ `|`. `SHOW_ALL` – это специальное значение, в котором установлены все биты: оно указывает, что должны рассматриваться все узлы независимо от их типа.

```
unsigned long SHOW_ALL = 0xFFFFFFFF;
unsigned long SHOW_ELEMENT = 0x00000001;
unsigned long SHOW_ATTRIBUTE = 0x00000002;
unsigned long SHOW_TEXT = 0x00000004;
unsigned long SHOW_CDATA_SECTION = 0x00000008;
unsigned long SHOW_ENTITY_REFERENCE = 0x00000010;
unsigned long SHOW_ENTITY = 0x00000020;
unsigned long SHOW_PROCESSING_INSTRUCTION = 0x00000040;
unsigned long SHOW_COMMENT = 0x00000080;
unsigned long SHOW_DOCUMENT = 0x00000100;
unsigned long SHOW_DOCUMENT_TYPE = 0x00000200;
unsigned long SHOW_DOCUMENT_FRAGMENT = 0x00000400;
unsigned long SHOW_NOTATION = 0x00000800;
```

Методы

`acceptNode()`

В языках, подобных Java, не допускающих передачи функций в качестве аргументов, фильтр узлов указывается путем определения класса, который реализует этот интерфейс и включает реализацию этой функции. Функции передается узел, и она должна вернуть одну из констант: `FILTER_ACCEPT`, `FILTER_REJECT` или `FILTER_SKIP`. Однако в JavaScript программист создает фильтр узлов, определяя функцию (с любым именем), принимающую аргумент-узел и возвращающую одну из трех констант фильтрации. Подробности и пример см. в последующих разделах.

Описание

Фильтр узлов – это объект, который может просматривать узел документа и сообщать NodeIterator или TreeWalker, надо ли включать узел в обход документа. В JavaScript фильтр узлов представляет собой функцию, принимающую один узел-аргумент и возвращающую одну из трех констант `FILTER_`, определенных ранее. Интерфейс NodeFilter в JavaScript отсутствует; имеется просто объект с именем NodeFilter и свойствами, определяющими эти константы. Для того чтобы фильтр узлов работал, надо передать этот объект методу `createNodeIterator()` или `createTreeWalker()` объекта Document. Ваша функция фильтрации узлов будет затем вызываться при использовании полученного объекта NodeIterator или TreeWalker для обхода документа.

В идеальном случае функции фильтрации узлов следует писать таким образом, чтобы они сами не изменяли дерево документа и не генерировали никаких исключений. Кроме того, функциям фильтрации узлов не разрешается основывать свое решение о фильтрации на истории предыдущих вызовов этих фильтров.

Пример

Функцию фильтрации узлов можно определить и использовать следующим образом:

```
// Определяем фильтр узлов, который отфильтровывает все, кроме элементов <h1> и <h2>
var myfilter = function(n) {       // Фильтруемый узел n
    if ((n.nodeName == "H1") || (n.nodeName == "H2"))
        return NodeFilter.FILTER_ACCEPT;
    else
        return NodeFilter.FILTER_SKIP;
}

// Теперь создаем NodeIterator, использующий этот фильтр
var ni = document.createNodeIterator(document.body,    // Обходим тело документа
                                     NodeFilter.SHOW_ELEMENT,   // Только элементы
                                     myfilter,  // Фильтрует по имени тега
                                     false);    // Сущности не разворачиваем
```

См. также

NodeIterator, TreeWalker

Тип: NodeIterator.filter, TreeWalker.filter

Передается в: Document.createNodeIterator(), Document.createTreeWalker()

NodeIterator

DOM Level 2 Traversal

обход фильтрованной последовательности узлов документа

Свойства

`readonly boolean expandEntityReferences`

Обходит ли этот NodeIterator дочерние узлы узлов EntityReference (в XML-документах). Значение задается в виде аргумента метода `Document.createNodeIterator()`, когда NodeIterator создается в первый раз.

`readonly NodeFilter filter`

Функция фильтрации узлов, которая была задана для этого NodeIterator при вызове `Document.createNodeIterator()`.

`readonly Node root`

Корневой узел, от которого NodeIterator начинает обход. Значение этого свойства задается при вызове `Document.createNodeIterator()`.

`readonly unsigned long whatToShow`

Набор битовых флагов (список допустимых флагов см. в «NodeFilter»), указывающих, какие типы узлов документа будет рассматривать этот NodeIterator. Если в этом свойстве бит не установлен, соответствующий тип узлов будет всегда игнорироваться этим NodeIterator. Обратите внимание, что значение этого свойства задается при вызове `Document.createNodeIterator()`.

Методы

`detach()`

«Отсоединяет» этот NodeIterator от данного документа так, что реализации более не требуется модифицировать NodeIterator при модификации документа. Этот метод надо вызывать, закончив использование NodeIterator. После вызова `detach()` любые вызовы других методов NodeIterator будут приводить к исключениям.

`nextNode()`

Возвращает следующий узел в фильтрованной последовательности узлов, представляемых данным NodeIterator, или `null`, если NodeIterator уже вернул последний узел.

`previousNode()`

Возвращает предыдущий узел в фильтрованной последовательности узлов, представляемых данным NodeIterator, или `null`, если предыдущий узел отсутствует.

Описание

NodeIterator представляет последовательность узлов документа, полученных в результате обхода поддерева документа в порядке их следования в исходном тексте и двухстадийного процесса фильтрации узлов. Объект NodeIterator создается при помощи метода `Document.createNodeIterator()`. Методы `nextNode()` и `previousNode()` позволяют перемещаться вперед и назад по последовательности узлов. Закончив работу с NodeIterator, вызовите `detach()`, если только не уверены, что NodeIterator будет уничтожен сборщиком мусора до того, как документ будет изменен. Обратите внимание, что свойства этого интерфейса являются доступными только для чтения копиями аргументов, переданных в `Document.createNodeIterator()`.

Для того чтобы методы `nextNode()` и `previousNode()` возвращали узел, он должен пройти два этапа фильтрации. Во-первых, узел должен относиться к одному из типов, заданных в свойстве `whatToShow`. Список констант, которые могут объединяться для задания значения аргумента *`чтоПоказывать`* метода `Document.createNodeIterator()`, см. в статье «NodeFilter». Во-вторых, если свойство `filter` не равно `null`, каждый узел, прошедший проверку `whatToShow`, передается в функцию фильтрации, заданную свойством `filter`. Если эта функция возвращает `NodeFilter.FILTER_ACCEPT`, узел возвращается. Если она возвращает `NodeFilter.FILTER_REJECT` или `NodeFilter.FILTER_SKIP`, NodeIterator пропускает этот узел. Обратите внимание, что когда узел отвергается одним из этих этапов фильтрации, отвергается только сам узел; дочерние узлы не отвергаются автоматически и проходят те же этапы фильтрации.

Объекты NodeIterator остаются действующими, даже если дерево документа, которое они обходят, изменяется. Методы `nextNode()` и `previousNode()` возвращают узлы, основываясь на текущем состоянии документа, а не на состоянии документа на момент создания NodeIterator.

См. также

NodeFilter, TreeWalker;

Возвращается из: Document.createNodeIterator()

NodeIterator.detach() DOM Level 2 Traversal

освобождает объект NodeIterator

Синтаксис

```
void detach();
```

Описание

Реализации DOM отслеживают все объекты NodeIterator, созданные для документа, так как может потребоваться изменить состояние NodeIterator при удалении узлов документа. Если вы уверены, что NodeIterator больше не потребуется, вызовите `detach()`, чтобы сообщить реализации, что его состояние больше не надо отслеживать. Заметьте, однако, что после вызова этого метода все последующие вызовы `nextNode()` или `previousNode()` будут генерировать исключение.

Вызов `detach()` не является обязательным, но может улучшить производительность, когда документ изменяется, а объект NodeIterator не уничтожается немедленно сборщиком мусора.

NodeIterator.nextNode() DOM Level 2 Traversal

переходит к следующему узлу

Синтаксис

```
Node nextNode()
    throws DOMException;
```

Возвращаемое значение

Следующий узел в последовательности узлов, представляемых этим NodeIterator, или `null`, если уже был получен последний узел.

Исключения

Если этот метод вызывается после вызова `detach()`, то он генерирует исключение DOMException с кодом `INVALID_STATE_ERR`.

Описание

Этот метод перемещается вперед по последовательности узлов, представляемых этим NodeIterator. Если этот метод вызывается для NodeIterator в первый раз, он возвращает первый узел в последовательности. В противном случае он возвращает узел, следующий за полученным ранее.

Пример

```
// Создаем NodeIterator для представления всех элементов в теле документа
var ni = document.createNodeIterator(document.body, NodeFilter.SHOW_ELEMENT,
                                     null, false);
// Цикл вперед по всем узлам в итераторе
for(var e = ni.nextNode(); e != null; e = ni.nextNode()) {
    // Выполняем какие-либо действия с элементом e
}
```

NodeIterator.previousNode() DOM Level 2 Traversal

переход к предыдущему узлу

Синтаксис

```
Node previousNode()
    throws DOMException;
```

Возвращаемое значение

Предыдущий узел в последовательности узлов, представляемых этим объектом NodeIterator, или `null`, если предыдущий узел отсутствует.

Исключения

Если этот метод вызывается после вызова `detach()`, то он генерирует исключение DOMException с кодом `INVALID_STATE_ERR`.

Описание

Этот метод обходит последовательность, представляемую данным NodeIterator, в обратном порядке. Он возвращает узел перед тем узлом, который был получен последним с помощью `previousNode()` или `nextNode()`. Если такого узла в последовательности нет, метод возвращает `null`.

NodeList DOM Level 1 Core

массив узлов, доступный только для чтения

Свойства

`readonly unsigned long length`

Количество узлов в массиве.

Методы

`item()`

Возвращает указанный элемент массива.

Описание

Интерфейс NodeList определяет доступный только для чтения упорядоченный список (то есть массив) объектов Node. Свойство `length` указывает, сколько узлов находится в списке, а метод `item()` позволяет получить узел в указанной позиции в списке. Элементы NodeList всегда являются корректными объектами Node: NodeList никогда не содержит пустых (`null`) элементов.

В JavaScript объекты NodeList ведут себя так же, как массивы JavaScript, и элемент из списка можно получить, используя нотацию массивов с квадратными скобками вместо вызова метода `item()`. Однако нельзя присваивать новые узлы NodeList с помощью квадратных скобок. Поскольку всегда проще рассматривать объект NodeList как доступный только для чтения массив JavaScript, в этой книге используется нотация `Node[]` (то есть массив Node) вместо NodeList. Обратите, например, внимание на метод «Element.getElementsByTagName()», который описан как возвращающий `Node[]`, а не NodeList. Аналогично, свойство `childNodes` объекта Node формально является объектом NodeList, но на справочной странице «Node» оно определяется как `Node[]`, а само свойство обычно упоминается как «массив `childNodes[]`».

Обратите внимание, что объекты NodeList «живые»: они немедленно отражают изменения в дереве документа. Например, если NodeList представляет дочерние узлы для указанного узла, и вы удалите один из этих дочерних узлов, он будет удален и из вашего NodeList. Будьте аккуратны при выполнении цикла по элементам NodeList, если тело цикла вносит изменения в дерево документа (например, удаляет узлы), которые могут влиять на содержимое NodeList!

См. также

NamedNodeMap

Тип: Node.childNodes

Возвращается из: Document.getElementsByTagName(), Document. getElementsByTagNameNS(), Element.getElementsByTagName(), Element. getElementsByTagNameNS(), HTMLDocument.getElementsByName()

NodeList.item()

DOM Level 1 Core

получает элемент из NodeList

Синтаксис

```
Node item(unsigned long индекс);
```

Аргументы

индекс

Позиция (индекс) нужного узла в NodeList. Индекс первого узла в NodeList равен 0, а индекс последнего узла равен `length` –1.

Возвращаемое значение

Узел в указанной позиции в NodeList или `null`, если *индекс* меньше нуля или больше или равен длине NodeList.

Описание

Этот метод возвращает указанный элемент NodeList. В JavaScript вместо вызова `item()` можно использовать нотацию массива с квадратными скобками.

Notation

DOM Level 1 XML

нотация в XML DTD **Node → Notation**

Свойства

`readonly String publicId`

Публичный идентификатор нотации или `null`, если он отсутствует.

`readonly String systemId`

Системный идентификатор нотации или `null`, если он отсутствует.

Описание

Этот редко используемый интерфейс представляет объявление нотации в определении типа документа (DTD) для документа XML. В XML нотации применяются для задания формата необработанной сущности или формального объявления цели инструкции обработки.

Имя нотации задается унаследованным свойством `nodeName`. Нотации присутствуют в DTD, а не в самом документе, поэтому узлы Notation никогда не являются частью дерева документа, а свойство `parentNode` всегда равно `null`. Свойство `notations` интерфейса DocumentType предоставляет способ поиска объектов Notation по имени нотации.

Объекты Notation доступны только для чтения и не могут каким-либо образом модифицироваться.

См. также

DocumentType

ProcessingInstruction

DOM Level 1 XML

инструкция обработки в XML-документе **Node → ProcessingInstruction**

Свойства

`String data`

Содержимое инструкции обработки (то есть от первого непробельного символа после цели до закрывающих символов `?>`, но не включая их).

`readonly String target`

Цель инструкции обработки. Это первый идентификатор инструкции обработки, следующий за открывающими символами `<?`; он задает «обработчик», для которого предназначена инструкция обработки.

Описание

Этот редко применяемый интерфейс представляет инструкцию обработки в документе XML. Программисты, работающие с HTML-документами, никогда не столкнутся с узлом ProcessingInstruction.

См. также

Возвращается из: Document.createProcessingInstruction()

Range

DOM Level 2 Range

представляет непрерывную область документа

Константы

Эти константы задают, как должны сравниваться граничные точки двух объектов Range. Они являются допустимыми значениями аргумента *какСравнивать* в методе `compareBoundaryPoints()`. См. справочную страницу «Range.compareBoundaryPoints()».

`unsigned short START_TO_START = 0`

Сравнивает начало указанной области с началом этой области.

`unsigned short START_TO_END = 1`

Сравнивает начало указанной области с концом этой области.

`unsigned short END_TO_END = 2`

Сравнивает конец указанной области с концом этой области.

`unsigned short END_TO_START = 3`

Сравнивает конец указанной области с началом этой области.

Свойства

Интерфейс Range определяет следующие свойства. Обратите внимание, что все эти свойства доступны только для чтения. Нельзя изменить начальную и конечную точки области путем установки свойств; вместо этого необходимо вызывать методы `setEnd()` или `setStart()`. Обратите также внимание, что после вызова метода `detach()` объекта Range любые последующие попытки прочитать любые из этих свойств генерируют исключение DOMException с кодом `INVALID_STATE_ERR`.

`readonly boolean collapsed`

`true`, если начало и конец области находятся в одной точке документа, то есть область пуста или «сжата».

`readonly Node commonAncestorContainer`

Наиболее глубоко вложенный узел документа, содержащий (то есть являющийся предком) как начальную, так и конечную точки области.

`readonly Node endContainer`

Узел документа, содержащий конечную точку области.

`readonly long endOffset`

Позиция конечной точки внутри `endContainer`.

`readonly Node startContainer`

Узел документа, содержащий начальную точку области.

`readonly long startOffset`

Позиция начальной точки области внутри `startContainer`.

Методы

Интерфейс Range определяет следующие методы. Обратите внимание, что если метод `detach()` вызван для области, то любые последующие вызовы любых методов для этой области генерируют исключение DOMException с кодом `INVALID_STATE_ERR`. Так как это исключение повсеместно встречается в данном интерфейсе, оно не перечисляется на справочных страницах отдельных методов Range.

`cloneContents()`

Возвращает новый объект DocumentFragment, содержащий копию области документа, представляемой данным объектом Range.

`cloneRange()`

Создает новый объект Range, представляющий такую же область документа, как данный объект.

`collapse()`

Сжимает область так, что одна граничная точка становится такой же, как другая.

`compareBoundaryPoints()`

Сравнивает граничную точку указанной области с граничной точкой данной области и возвращает –1, 0 или 1, в зависимости от их порядка. Первый аргумент задает, какие точки должны сравниваться, и его значение должно быть одной из ранее определенных констант.

`deleteContents()`

Удаляет область документа, представляемую данным объектом Range.

`detach()`

Сообщает реализации, что эта область больше не будет использоваться и можно прекратить отслеживание изменений в ней. Если вызвать этот метод для области, последующие вызовы или обращения к свойствам этой области будут генерировать исключение DOMException с кодом `INVALID_STATE_ERR`.

`extractContents()`

Удаляет область документа, представленную данным объектом Range, но возвращает содержимое этой области в виде объекта DocumentFragment. Действие этого метода равно сумме действий методов `cloneContents()` и `deleteContents()`.

`insertNode()`

Вставляет указанный узел в документ в качестве начальной точки области.

`selectNode()`

Устанавливает граничные точки этой области так, чтобы она содержала указанный узел и всех его потомков.

`selectNodeContents()`

Устанавливает граничные точки этой области так, чтобы она содержала всех потомков указанного узла, но не сам узел.

`setEnd()`

Устанавливает конечную точку этой области в указанном узле и смещении.

`setEndAfter()`

Устанавливает конечную точку области непосредственно после указанного узла.

`setEndBefore()`

Устанавливает конечную точку области непосредственно перед указанным узлом.

`setStart()`

Устанавливает начальную позицию области с указанным смещением внутри указанного узла.

`setStartAfter()`

Устанавливает начальную позицию этой области непосредственно после указанного узла.

`setStartBefore()`

Устанавливает начальную позицию этой области непосредственно перед указанным узлом.

`surroundContents()`

Вставляет указанный узел в документ в начальную позицию области и затем меняет родителей всех узлов в области таким образом, что они становятся потомками только что вставленного узла.

`toString()`

Возвращает текстовое содержимое области документа, описываемой этим объектом Range.

Описание

Объект Range представляет непрерывную область документа, как, например, область, которую пользователь может выбрать перетаскиванием в окне веб-броузера.

Если реализация поддерживает модуль Range, объект Document определяет метод `createRange()`, который можно вызвать для создания нового объекта Range. (Однако будьте осторожны: Internet Explorer определяет несовместимый метод `Document.createRange()`, возвращающий объект, аналогичный, но не совместимый с интерфейсом Range.) Интерфейс Range определяет несколько методов для задания «выбранной» области документа и еще несколько методов для реализации операций вставки-замены в выбранной области.

Область имеет две граничные точки: начальную и конечную. Каждая граничная точка задается комбинацией узла и смещения внутри узла. Узел – это обычно Element, Document или Text. Для узлов Element и Document смещение относится к дочерним элементам узла. Смещение 0 определяет граничную точку перед первым дочерним узлом данного. Смещение 1 определяет граничную точку после первого дочернего узла и перед вторым дочерним узлом. Однако если граничный узел – это узел Text, то смещение определяет позицию между двумя символами этого текста.

Свойства интерфейса Range предоставляют способ получения граничных узлов и смещений области. Методы интерфейса предоставляют несколько способов установки границ области. Обратите внимание, что границы области могут быть установлены в узлах внутри документа или внутри DocumentFragment.

Определив граничные точки области, можно вызывать методы `deleteContents()`, `extractContents()`, `cloneContents()` и `insertNode()` для реализации операций вырезания, копирования и вставки.

Когда документ изменяется путем вставки или удаления узлов, все объекты Range, представляющие измененные части документа, по необходимости изменяются, чтобы их граничные точки оставались действующими и представляли (насколько это возможно) то же содержимое документа.

Подробности см. на справочных страницах для каждого из методов Range, а также в обсуждении Range API в главе 17.

См. также

Document.createRange(), DocumentFragment; главу 17

Передается в: Range.compareBoundaryPoints()

Возвращается из: Document.createRange(), Range.cloneRange()

Range.cloneContents() DOM Level 2 Range

копирует содержимое области в DocumentFragment

Синтаксис

```
DocumentFragment cloneContents()
    throws DOMException;
```

Возвращаемое значение

Объект DocumentFragment, заключающий в себе копию содержимого документа внутри этой области.

Исключения

Если эта область включает узел DocumentType, то данный метод генерирует исключение DOMException с кодом `HIERARCHY_REQUEST_ERR`.

Описание

Этот метод дублирует содержимое области и возвращает результат в объекте DocumentFragment.

См. также

DocumentFragment, Range.deleteContents(), Range.extractContents()

Range.cloneRange() DOM Level 2 Range

создает копию этой области

Синтаксис

```
Range cloneRange();
```

Возвращаемое значение

Новый объект Range, имеющий те же граничные точки, что и эта область.

См. также

Document.createRange()

Range.collapse() DOM Level 2 Range

делает одну граничную точку равной другой

Синтаксис

```
void collapse(boolean вНачало)
    throws DOMException;
```

Аргументы

вНачало

Если этот аргумент равен `true`, метод устанавливает конечную точку области равной начальной точке. В противном случае он устанавливает начальную точку равной конечной точке.

Описание

Этот метод устанавливает одну граничную точку области равной другой точке. Модифицируемая точка задается аргументом *вНачало*. После завершения метода про область говорят, что она «сжатая»: она представляет одну точку в документе и не имеет содержимого. Если область сжата подобным образом, свойство `collapsed` равно `true`.

Range.compareBoundaryPoints() DOM Level 2 Range

сравнивает позиции двух областей

Синтаксис

```
short compareBoundaryPoints(unsigned short какСравнивать,
                            Range исходнаяОбласть)
    throws DOMException;
```

Аргументы

какСравнивать

Указывает, как выполнять сравнение (то есть сравниваемые граничные точки). Допустимые значения представляют собой константы, определенные в интерфейсе Range.

исходнаяОбласть

Область, которая должна сравниваться с данной.

Возвращаемое значение

–1, если указанная граничная точка области находится перед указанной граничной точкой *исходнойОбласти*, 0, если две указанные граничные точки совпадают, или 1, если указанная граничная точка находится после указанной граничной точки *исходнойОбласти*.

Исключения

Если *исходнаяОбласть* представляет собой область другого документа, этот метод генерирует исключение DOMException с кодом `WRONG_DOCUMENT_ERR`.

Описание

Этот метод сравнивает граничную точку области с граничной точкой указанной *исходнойОбласти* и возвращает значение, задающее их относительный порядок в документе. Аргумент *какСравнивать* задает, какие граничные точки каждой из областей должны сравниваться. Ниже приведены допустимые значения этого аргумента и объяснен их смысл:

`Range.START_TO_START`

Сравнивает начальные точки двух объектов Range.

`Range.END_TO_END`

Сравнивает конечные точки двух объектов Range.

`Range.START_TO_END`

Сравнивает начальную точку *исходнойОбласти* с конечной точкой этой области.

`Range.END_TO_START`

Сравнивает конечную точку *исходнойОбласти* с начальной точкой этой области.

Возвращаемое значение данного метода – число, задающее относительную позицию этой области по отношению к указанной *исходнойОбласти*. Следовательно, можно предполагать, что константы для аргумента *какСравнивать* будут определять граничную точку данной области как первую, а граничную область *исходнойОбласти* как вторую. Но, вопреки здравому смыслу, константа `Range.START_TO_END` задает сравнение ***конечной*** точки данной области с ***начальной*** точкой указанной *исходнойОбласти*. Аналогично, константа `Range.END_TO_START` задает сравнение ***начальной*** точки данной области с ***конечной*** точкой указанной *исходнойОбласти*.

Range.deleteContents()

DOM Level 2 Range

удаляет область из документа

Синтаксис

```
void deleteContents()
    throws DOMException;
```

Исключения

Если какая-либо из частей документа, представляемая этой областью, доступна только для чтения, данный метод генерирует исключение DOMException с кодом, равным `NO_MODIFICATION_ALLOWED_ERR`.

Описание

Метод удаляет все содержимое документа, заданное этой областью. Когда он завершает работу, область сжимается, и обе граничные точки оказываются в одной позиции. Обратите внимание, что удаление может приводить к созданию смежных узлов Text, которые могут быть объединены с помощью метода `Node.normalize()`.

Способ копирования содержимого см. в описании метода «cloneContents()», а способ копирования и удаления содержимого документа как единая операция описан в статье «extractContents()».

См. также

Node.normalize(), Range.cloneContents(), Range.extractContents()

Range.detach() DOM Level 2 Range

освобождает объект Range

Синтаксис

```
void detach()
    throws DOMException;
```

Исключения

Как все методы Range, `detach()` генерирует исключение DOMException с кодом `INVALID_STATE_ERR`, если он вызывается для объекта Range, который уже был отсоединен.

Описание

Реализации DOM отслеживают все объекты Range, созданные для документа, так как при изменении документа может потребоваться изменить граничные точки области. Если вы уверены, что объект Range больше не потребуется, вызовите метод `detach()`, чтобы сообщить реализации, что эту область больше не надо отслеживать. Обратите внимание, что после вызова этого метода для объекта Range любое использование этого объекта будет генерировать исключение. Вызов `detach()` не является обязательным, но может улучшить производительность, когда документ изменяется, а объект Range не уничтожается немедленно сборщиком мусора.

Range.extractContents() DOM Level 2 Range

удаляет содержимое документа и возвращает его в виде DocumentFragment

Синтаксис

```
DocumentFragment extractContents()
    throws DOMException;
```

Возвращаемое значение

Узел DocumentFragment, в котором находится содержимое этой области.

Исключения

Этот метод генерирует исключение DOMException с кодом `NO_MODIFICATION_ALLOWED_ERR`, если какая-либо часть извлекаемого содержимого документа доступна только для чтения, или с кодом `HIERARCHY_REQUEST_ERR`, если область содержит узел DocumentType.

Описание

Метод удаляет указанную область документа и возвращает узел DocumentFragment, содержащий удаленное содержимое (или копию удаленного содержимого). Когда этот метод завершает свою работу, область сжимается, и документ может содержать смежные узлы Text (которые могут быть объединены с помощью `Node.normalize()`).

См. также

DocumentFragment, Range.cloneContents(), Range.deleteContents()

Range.insertNode() DOM Level 2 Range

вставляет узел в начало области

Синтаксис

```
void insertNode(Node новыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

новыйУзел

Узел, который должен быть вставлен в документ.

Исключения

Этот метод генерирует исключение RangeException с кодом `INVALID_NODE_TYPE_ERR`, если *новыйУзел* является узлом Attr, Document, Entity или Notation.

Он также генерирует исключение DOMException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`HIERARCHY_REQUEST_ERR`

Узел, содержащий начало области, не допускает наличия дочерних узлов или дочерних узлов указанного типа, или *новыйУзел* является предком данного узла.

`NO_MODIFICATION_ALLOWED_ERR`

Узел, содержащий начало области, или любые его предки доступны только для чтения.

`WRONG_DOCUMENT_ERR`

`новыйУзел` – это часть документа, отличного от данного.

Описание

Метод вставляет указанный узел (и всех его потомков) в документ в начальной позиции этой области. Когда данный метод завершает работу, эта область включает только вставленный узел. Если *новыйУзел* уже является частью документа, он удаляется из своей текущей позиции и вставляется заново в начале области. Если *новыйУзел* является узлом DocumentFragment, он не вставляется сам, а в начало области вставляются по порядку все его дочерние элементы.

Если узел, содержащий начало области, – это узел Text, то перед вставкой он разбивается на два смежных узла. Если *новыйУзел* – это узел Text, то он после вставки не объединяется с любыми смежными узлами Text. Для объединения смежных узлов следует применять `Node.normalize()`.

См. также

DocumentFragment, Node.normalize()

Range.selectNode() DOM Level 2 Range

ограничивает область указанным узлом

Синтаксис

```
void selectNode(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, который должен быть «выбран» (то есть узел, который должен стать содержимым данной области).

Исключения

RangeException с кодом `INVALID_NODE_TYPE_ERR`, если выбираемый узел является узлом Attr, Document, DocumentFragment, Entity или Notation или если любой предок *выбираемогоУзла* является узлом DocumentType, Entity или Notation.

DOMException с кодом, равным `WRONG_DOCUMENT_ERR`, если *выбираемыйУзел* является частью документа, отличного от того, в котором был создан данный объект Range.

Описание

Этот метод устанавливает содержимое области равным указанному *выбираемомуУзлу*, то есть выбирает этот узел и всех его потомков.

См. также

Range.selectNodeContents()

Range.selectNodeContents() DOM Level 2 Range

устанавливает границы области в дочерних узлах данного узла

Синтаксис

```
void selectNodeContents(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, дочерние элементы которого становятся содержимым этой области.

Исключения

RangeException с кодом INVALID_NODE_TYPE_ERR, если выбираемый узел или один из его предков является узлом DocumentType, Entity или Notation.

DOMException с кодом, равным WRONG_DOCUMENT_ERR, если *выбираемыйУзел* является частью документа, отличного от того, с помощью которого был создан данный объект Range.

Описание

Этот метод устанавливает граничные точки этой области так, что область содержит дочерние узлы *выбираемогоУзла*.

См. также

Range.selectNode()

Range.setEnd() DOM Level 2 Range

устанавливает конечную точку области

Синтаксис

```
void setEnd(Node выбираемыйУзел,
            long смещение)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, содержащий новую конечную точку.

смещение

Позиция конечной точки внутри *выбираемогоУзла*.

Исключения

RangeException с кодом INVALID_NODE_TYPE_ERR, если выбираемый узел или один из его предков является узлом DocumentType, Entity или Notation.

DOMException с кодом, равным WRONG_DOCUMENT_ERR, если *выбираемыйУзел* представляет собой часть документа, отличного от того, с помощью которого был создан данный объект Range, или с кодом INDEX_SIZE_ERR, если *смещение* отрицательно или больше количества дочерних узлов или символов в *выбираемомУзле*.

Описание

Этот метод устанавливает конечную точку области путем задания значений свойств endContainer и endOffset.

Range.setEndAfter() DOM Level 2 Range

завершает область после указанного узла

Синтаксис

```
void setEndAfter(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, после которого должна быть установлена конечная точка области.

Исключения

RangeException с кодом `INVALID_NODE_TYPE_ERR`, если *выбираемыйУзел* является узлом Document, DocumentFragment, Attr, Entity или Notation или если корневой контейнер *выбираемогоУзла* не является узлом Document, DocumentFragment или Attr.

DOMException с кодом, равным `WRONG_DOCUMENT_ERR`, если *выбираемыйУзел* представляет собой часть документа, отличного от того, с помощью которого был создан данный объект Range.

Описание

Метод устанавливает конечную точку этой области непосредственно после указанного *выбираемогоУзла*.

Range.setEndBefore() DOM Level 2 Range

завершает область перед указанным узлом

Синтаксис

```
void setEndBefore(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, перед которым должна быть установлена конечная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и `Range.setEndAfter()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает конечную точку данной области непосредственно перед *выбираемымУзлом*.

Range.setStart() DOM Level 2 Range

устанавливает начальную точку области

Синтаксис

```
void setStart(Node выбираемыйУзел,
              long смещение)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, содержащий новую начальную точку.

смещение

Позиция новой начальной точки внутри *выбираемогоУзла*.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и `Range.setEnd()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области путем задания значений свойств `startContainer` и `startOffset`.

Range.setStartAfter() DOM Level 2 Range

начинает область непосредственно после указанного узла

Синтаксис

```
void setStartAfter(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, после которого должна быть установлена начальная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и `Range.setEndAfter()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области непосредственно после указанного *выбираемогоУзла*.

Range.setStartBefore() DOM Level 2 Range

начинает область перед указанным узлом

Синтаксис

```
void setStartBefore(Node выбираемыйУзел)
    throws RangeException,
           DOMException;
```

Аргументы

выбираемыйУзел

Узел, перед которым должна быть установлена начальная точка области.

Исключения

Этот метод генерирует те же исключения и при тех же обстоятельствах, что и `Range.setEndAfter()`. Подробности см. в описании этого метода.

Описание

Метод устанавливает начальную точку данной области непосредственно перед указанным *выбираемымУзлом*.

Range.surroundContents()

DOM Level 2 Range

окружает содержимое области указанным узлом

Синтаксис

```
void surroundContents(Node новыйРодитель)
    throws RangeException,
           DOMException;
```

Аргументы

новыйРодитель

Узел, становящийся новым родительским узлом для содержимого данной области.

Исключения

Этот метод генерирует исключение DOMException или RangeException с одним из перечисленных ниже значений `code` в следующих ситуациях:

`DOMException.HIERARCHY_REQUEST_ERR`

Узел-контейнер начала области не допускает наличия дочерних узлов или дочерних узлов типа узла *новыйРодитель*, или *новыйРодитель* является предком данного узла-контейнера.

`DOMException.NO_MODIFICATION_ALLOWED_ERR`

Предок граничной точки области доступен только для чтения и не допускает вставок.

`DOMException.WRONG_DOCUMENT_ERR`

новыйРодитель и эта область созданы с помощью разных объектов Document.

`RangeException.BAD_BOUNDARYPOINTS_ERR`

Область частично выбирает узел (отличный от узла Text), так что представляемая этим объектом область документа не может быть окружена узлом.

`RangeException.INVALID_NODE_TYPE_ERR`

новыйРодитель – это узел Document, DocumentFragment, DocumentType, Attr, Entity или Notation.

Описание

Этот метод меняет родителя для содержимого этой области на узел *новыйРодитель* и затем вставляет этот узел в документ в начальной позиции области. Это, например, полезно для помещения области документа внутрь элемента `<b>` или `<span>`.

Если *новыйРодитель* уже является частью документа, он сначала удаляется из документа и все его дочерние элементы уничтожаются. После завершения работы метода область начинается непосредственно перед *новымРодителем*, а заканчивается сразу после него.

Range.toString()

DOM Level 2 Range

извлекает содержимое области в виде строки текста

Синтаксис

```
String toString();
```

Возвращаемое значение

Содержимое этой области в виде строки обычного текста без какой-либо разметки.

RangeException

DOM Level 2 Range

сигнализирует об исключении, относящемся к Range API

Константы

Следующие константы определяют допустимые значения свойства `code` объекта RangeException. Обратите внимание, что эти константы представляют собой статические свойства RangeException, а не свойства индивидуальных объектов исключений.

`unsigned short BAD_BOUNDARYPOINTS_ERR = 1`

Граничные точки области не допустимы для данной операции.

`unsigned short INVALID_NODE_TYPE_ERR = 2`

Была предпринята попытка установить узел-контейнер граничной точки области равным недопустимому узлу или узлу с недопустимым предком.

Свойства

`unsigned short code`

Код ошибки, предоставляющий сведения о причинах ошибки. Допустимые значения данного свойства (и их смысл) определяются только что перечисленными константами.

Описание

RangeException генерируется определенными методами интерфейса Range, сигнализируя о какой-либо проблеме. Обратите внимание, что большинство исключений, генерируемых методами Range, представляют собой объекты DOMException. RangeException генерируется, только если ни одна из существующих констант ошибок DOMException не подходит для описания исключения.

Rect

DOM Level 2 CSS

значение CSS rect()

Свойства

`readonly CSSPrimitiveValue bottom`

Нижний край прямоугольника.

`readonly CSSPrimitiveValue left`

Левая сторона прямоугольника.

`readonly CSSPrimitiveValue right`

Правая сторона прямоугольника.

`readonly CSSPrimitiveValue top`

Нижний край прямоугольника.

Описание

Этот интерфейс представляет значение `rect(top right bottom left)`, используемое, например, в CSS-атрибуте `clip`. Подробности см. в литературе по CSS.

См. также

Возвращается из: CSSPrimitiveValue.getRectValue()

RGBColor

DOM Level 2 CSS

цветовое значение CSS

Свойства

`readonly CSSPrimitiveValue blue`

Синяя компонента цвета.

`readonly CSSPrimitiveValue green`

Зеленая компонента цвета.

`readonly CSSPrimitiveValue red`

Красная компонента цвета.

Описание

Это интерфейс, который представляет цвет, определяемый в цветовом пространстве RGB. Свойства являются объектами CSSPrimitiveValue, задающими красную, зеленую и синюю компоненты цвета. В каждом CSSPrimitiveValue содержится число в диапазоне от 0 до 255 или процентное значение в диапазоне от 0 до 100%.

См. также

Возвращается из: CSSPrimitiveValue.getRGBColorValue()

StyleSheet

DOM Level 2 StyleSheets

таблица стилей любого типа

Подынтерфейсы

CSSStyleSheet

Свойства

`boolean disabled`

Если свойство равно `true`, таблица стилей отключена и не применяется к документу. Если оно равно `false`, таблица стилей включена и применяется к документу (если только свойство `media` не указывает, что стилевая таблица не должна применяться к документам этого типа).

`readonly String href`

URL таблицы стилей, на которую ссылается документ, или `null` для встроенной таблицы стилей.

`readonly MediaList media`

Список типов носителей, для которых должна применяться эта таблица стилей. Если информация о носителе для таблицы стилей не указана, то данное свойство представляет собой действительный объект MediaList, длина которого равна 0.

`readonly Node ownerNode`

Узел документа, который присоединяет таблицу стилей к документу, или узел, содержащий встроенную таблицу стилей. В документах HTML это свойство ссылается на элемент `<link>` или `<style>`. Для таблиц стилей, включаемых в другие таблицы стилей, а не в документ непосредственно, это свойство равно `null`.

`readonly StyleSheet parentStyleSheet`

Таблица стилей, включающая данную, или `null`, если эта таблица стилей была включена прямо в документ. См. также статью «CSSStyleSheet.ownerRule.»

`readonly String title`

Заголовок таблицы стилей, если он указан. Заголовок может быть задан атрибутом `title` тега `<style>` или `<link>`, на который ссылается свойство `ownerNode` этой таблицы стилей.

`readonly String type`

MIME-тип этой таблицы стилей. Таблица стилей CSS имеет тип «text/css».

Описание

Этот интерфейс представляет таблицу стилей, связанную с данным документом. Если это таблица стилей CSS, то объект, реализующий этот интерфейс StyleSheet, также реализует подынтерфейс CSSStyleSheet и определяет свойства и методы, позволяющие просматривать и устанавливать правила CSS, составляющие таблицу стилей. Подробности см. в статье «CSSStyleSheet».

Если реализация DOM поддерживает таблицу стилей, то полную таблицу стилей, связанную с документом, можно получить через свойство `Document.styleSheets`. Кроме того, элементы HTML `<style>` и `<link>` и XML-узлы ProcessingInstruction, ссылающиеся на таблицы стилей, реализуют интерфейс `LinkStyle` и предоставляют ссылку на StyleSheet через свойство с именем `sheet`.

См. также

CSSStyleSheet, Document.styleSheets, LinkStyle

Тип: LinkStyle.sheet, StyleSheet.parentStyleSheet

Возвращается из: StyleSheetList.item()

StyleSheetList

DOM Level 2 StyleSheets

массив таблиц стилей

Свойства

`readonly unsigned long length`

Количество объектов StyleSheet в массиве.

Методы

`item()`

Возвращает объект StyleSheet в указанной позиции в массиве или `null`, если указанная позиция отрицательна или больше или равна `length`.

Описание

Этот интерфейс определяет массив объектов StyleSheet. `length` задает количество таблиц стилей в массиве, а `item()` предоставляет способ получения таблицы стилей из указанной позиции. В JavaScript объект StyleSheetList можно рассматривать как доступный только для чтения массив и обращаться к нему с помощью обычной нотации массива с квадратными скобками, обходясь без вызова метода `item()`.

См. также

Тип: DocumentStyle.styleSheets

StyleSheetList.item() DOM Level 2 StyleSheets

обращение к массиву таблиц стилей

Синтаксис

```
StyleSheet item(unsigned long индекс);
```

Аргументы

индекс

Позиция таблицы стилей в массиве.

Возвращаемое значение

Объект StyleSheet в указанной позиции в массиве или `null`, если *индекс* отрицателен или больше или равен `length`. Обратите внимание, что в JavaScript обычно проще рассматривать объект StyleSheetList как массив и обращаться к нему не с помощью этого метода, а посредством нотации массивов с квадратными скобками.

Text DOM Level 1 Core

последовательность текста в HTML- или XML-документе **Node → CharacterData → Text**

Подынтерфейсы

CDATASection

Методы

`splitText()`

Разбивает данный узел Text на два в указанной символьной позиции и возвращает новый текстовый узел.

Описание

Узел Text представляет последовательность обычного текста в HTML- или XML-документе. Обычный текст находится внутри элементов HTML и XML и атрибутов, и узлы Text обычно являются дочерними по отношению к узлам Element и Attr. Узлы Text – это наследники CharacterData, а текстовое содержимое узла Text доступно через свойство `data`, унаследованное от CharacterData, или через свойство `nodeValue`, унаследованное от Node. Узлами Text можно манипулировать, применяя любые методы, унаследованные от CharacterData, или метод `splitText()`, определенный в самом интерфейсе Text. В узлах Text никогда не бывает дочерних узлов.

Способ удаления пустых узлов Text и объединения смежных узлов Text из поддерева документа см. в статье «Node.normalize()».

См. также

CharacterData, Node.normalize()

Возвращается из: Document.createTextNode(), Text.splitText()

Text.splitText() DOM Level 1 Core

разбивает узел Text на два

Синтаксис

```
Text splitText(unsigned long смещение)
    throws DOMException;
```

Аргументы

смещение

Символьная позиция, в которой нужно разбить узел Text.

Возвращаемое значение

Узел Text, выделенный из данного узла.

Исключения

Этот метод может генерировать исключение DOMException с одним из перечисленных ниже значений `code`:

`INDEX_SIZE_ERR`

смещение отрицательно или больше длины узла Text или Comment.

`NO_MODIFICATION_ALLOWED_ERR`

Узел доступен только для чтения и не может быть модифицирован.

Описание

Этот метод разбивает узел Text на два по указанному *смещению*. Исходный узел Text модифицируется так, чтобы он содержал весь текст до символа в позиции *смещение*, но не включая его. Создается новый узел, который содержит все символы от позиции *смещение* (и включая ее) до конца строки. Этот новый узел Text представляет собой значение, возвращаемое методом. Кроме того, если исходный узел Text имеет родительский узел, то новый узел вставляется в родительский непосредственно после исходного узла.

Интерфейс CDATASection является наследником Text, и его метод `splitText()` также может использоваться с узлами CDATASection, и в этом случае только что созданный узел будет узлом CDATASection, а не узлом Text.

См. также

Node.normalize()

TreeWalker

DOM Level 2 Traversal

обходит фильтрованное поддерево документа

Свойства

`Node currentNode`

Текущая позиция этого TreeWalker и узел, относительно которого работают все методы обхода объекта TreeWalker. Это узел, который был возвращен одним из этих методов обхода или, если ни один из этих методов пока не был вызван, совпадает со свойством `root`.

Обратите внимание, что это свойство доступно для чтения и записи, и его можно установить равным любому действительному узлу Document – даже тому, который не является потомком исходного узла `root`, или узлу, который был бы отвергнут фильтрами, используемыми данным TreeWalker. Если изменить значение этого свойства, то методы обхода начинают действовать относительно нового указанного узла. Попытка установить это свойство равным `null` генерирует исключение DOMException с кодом, равным `NOT_SUPPORTED_ERR`.

`readonly boolean expandEntityReferences`

Это доступное только для чтения свойство указывает, должен ли данный объект TreeWalker разворачивать ссылки на сущности, которые он встречает при обходе

документов XML. Значение этого свойства устанавливается при вызове `Document.createTreeWalker()`.

`readonly NodeFilter filter`

Функция фильтрации узлов, если она имеется, которая была задана для этого TreeWalker при вызове `Document.createTreeWalker()`. Если функция фильтрации узлов не используется, это свойство устанавливается равным `null`.

`readonly Node root`

Это доступное только для чтения свойство задает узел `root`, с которого TreeWalker начинает обход. Начальное значение свойства `currentNode` задается при вызове `Document.createTreeWalker()`.

`readonly unsigned long whatToShow`

Это доступное только для чтения свойство проедставляяет собой набор битовых флагов (список допустимых флагов см. в статье «NodeFilter»), задающий типы узлов Document, которые будет рассматривать данный TreeWalker. Если бит в этом свойстве не установлен, соответствующий тип узлов всегда будет игнорироваться данным TreeWalker. Обратите внимание, что значение этого свойства задается при вызове `Document.createTreeWalker()`.

Методы

`firstChild()`

Возвращает либо `null`, либо первый дочерний элемент текущего узла, который не отвергается фильтрами.

`lastChild()`

Возвращает либо `null`, либо последний дочерний узел текущего узла, который не отвергается фильтрами.

`nextNode()`

Возвращает либо `null`, либо следующий узел (в исходном тексте документ), который не отвергается фильтрами.

`nextSibling()`

Возвращает либо `null`, либо следующий смежный узел текущего узла, который не отвергается фильтрами.

`parentNode()`

Возвращает либо `null`, либо родительский узел или ближайшего предка текущего узла, который не отвергается фильтрами.

`previousNode()`

Возвращает либо `null`, либо предыдущий узел (в порядке исходного текста документа), который не отвергается фильтрами.

`previousSibling()`

Возвращает либо `null`, либо ближайший предыдущий смежный узел для текущего узла, который не отвергается фильтрами.

Описание

TreeWalker фильтрует указанное поддерево документа и определяет методы, позволяющие программам обходить фильтрованное дерево (структура которого может существенно отличаться от структуры исходного дерева). Объект TreeWalker создается

с помощью метода `createTreeWalker()` объекта Document. Создав TreeWalker, программист может вызывать его методы `firstChild()` и `nextSibling()` для обхода представляемого этим объектом фильтрованного поддерева так же, как он может использовать свойства `firstChild` и `nextSibling` интерфейса Node для обхода нефильтрованного дерева документа.

TreeWalker применяет те же два этапа фильтрации, что и NodeIterator. Различные методы обхода, определяемые в TreeWalker, будут возвращать только узлы, прошедшие оба этапа. Во-первых, тип узла должен соответствовать одному из типов, заданных свойством `whatToShow`. Список констант, которые могут объединяться для формирования аргумента *чтоПоказывать* в методе `Document.createTreeWalker()`, см. в статье «NodeFilter». Во-вторых, если свойство `filter` не равно `null`, каждый узел, прошедший проверку `whatToShow`, передается функции фильтрации, заданной свойством `filter`. Если эта функция возвращает `NodeFilter.FILTER_ACCEPT`, узел возвращается. Если она возвращает `NodeFilter.FILTER_REJECT`, TreeWalker пропускает узел и всех его потомков (в этом состоит отличие от фильтрации NodeIterator, при которой потомки никогда не отвергаются автоматически). Если функция фильтрации узлов возвращает `NodeFilter.FILTER_SKIP`, TreeWalker игнорирует узел, но рассматривает его потомков.

В отличие от NodeIterator, TreeWalker не модифицируется при модификации исходного документа. Текущий узел TreeWalker остается неизменным, даже если этот узел удаляется из документа. (И в этом случае TreeWalker может применяться для обхода дерева удаленных узлов, если такие имеются, окружающих текущий узел.)

Пример

```
// NodeFilter, отвергающий теги <font> и любые элементы с атрибутом
// class="sidebar" и любых потомков такого элемента
var filter = function(n) {
    if (n.nodeName == "FONT") return NodeFilter.FILTER_SKIP;
    if (n.nodeType == Node.ELEMENT_NODE && n.className == "sidebar")
        return NodeFilter.FILTER_REJECT;
    return NodeFilter.FILTER_ACCEPT;
}

// Создает TreeWalker, используя приведенный выше фильтр
var tw = document.createTreeWalker(document.body,  // Обходим тело HTML-документа
                                   // Рассматриваем все узлы, кроме комментариев
                                   ~NodeFilter.SHOW_COMMENT,
                                   filter,  // Используем приведенный выше фильтр
                                   false);  // Не разворачиваем ссылки на сущности

// Это рекурсивная функция, которая обходит документ, используя объект TreeWalker
function traverse(tw) {
    // Запоминаем текущий узел
    var currentNode = tw.currentNode;

    // Цикл по всем дочерним узлам текущего узла TreeWalker
    for(var c = tw.firstChild(); c != null; c = tw.nextSibling()) {
        process(c);    // Что-то делаем с дочерним узлом
        traverse(tw);  // Рекурсивно обрабатываем его дочерние узлы
    }

    // Возвращаем TreeWalker обратно к состоянию, в котором мы его нашли
    tw.currentNode = currentNode;
}
```

См. также

NodeFilter, NodeIterator; главу 17

Возвращается из: Document.createTreeWalker()

TreeWalker.firstChild() DOM Level 2 Traversal

возвращает первый дочерний узел, который не был отвергнут фильтрами

Синтаксис

```
Node firstChild();
```

Возвращаемое значение

Первый дочерний узел текущего угла, не отвергнутый фильтрами, или `null`, если такого дочернего узла нет.

Описание

Этот метод устанавливает `currentNode` на первый дочерний узел текущего узла, не отвергнутый фильтрами, и возвращает этот узел. Если такого дочернего узла нет, свойство `currentNode` остается неизменным и возвращается `null`.

TreeWalker.lastChild() DOM Level 2 Traversal

возвращает последний дочерний узел, который не был отвергнут фильтрами

Синтаксис

```
Node lastChild();
```

Возвращаемое значение

Последний дочерний узел текущего узла, не отвергнутый фильтрами, или `null`, если такого дочернего узла нет.

Описание

Этот метод устанавливает `currentNode` на последний дочерний узел текущего узла, не отвергнутый фильтрами, и возвращает этот узел. Если такого дочернего узла нет, свойство `currentNode` остается неизменным и возвращается `null`.

TreeWalker.nextNode() DOM Level 2 Traversal

возвращает следующий узел, не отвергнутый фильтрами

Синтаксис

```
Node nextNode();
```

Возвращаемое значение

Узел, следующий за текущим узлом в исходном тексте документа и не отвергнутый фильтрами, или `null`, если такой узел отсутствует.

Описание

Этот метод устанавливает `currentNode` на следующий узел (в порядке исходного текста документа), который не отвергнут фильтрами, и возвращает этот узел. Если такой узел отсутствует или если поиск следующего узла выводит TreeWalker из поддерева `root`, `currentNode` остается неизменным и метод возвращает `null`.

Обратите внимание, что этот метод «выпрямляет» древовидную структуру документа и возвращает узлы в том порядке, в котором они расположены в исходном тексте документа. Вызов `nextNode()` может привести к перемещению текущего узла вниз, вбок или вверх по дереву документа. Этот тип выпрямляющего обхода также выполняется с помощью `NodeIterator.nextNode()`.

TreeWalker.nextSibling() DOM Level 2 Traversal

возвращает следующий смежный узел, не отвергнутый фильтрами

Синтаксис

```
Node nextSibling();
```

Возвращаемое значение

Следующий смежный узел текущего узла, который не отвергнут фильтрами, или `null`, если такой узел отсутствует.

Описание

Этот метод устанавливает `currentNode` на следующий смежный узел текущего узла, который не отвергнут фильтрами, и возвращает этот смежный узел. Если такого узла нет, `currentNode` остается неизменным и возвращается `null`.

TreeWalker.parentNode() DOM Level 2 Traversal

возвращает ближайшего предка, который не отвергнут фильтрами

Синтаксис

```
Node parentNode();
```

Возвращаемое значение

Ближайший предок текущего узла, не отвергнутый фильтрами, или `null`, если такого предка нет.

Описание

Этот метод устанавливает `currentNode` на ближайшего предка текущего узла, который не отвергнут фильтрами, и возвращает этого предка. Если такого предка нет, свойство `currentNode` остается неизменным и возвращается `null`.

TreeWalker.previousNode() DOM Level 2 Traversal

возвращает предыдущий узел, не отвергнутый фильтрами

Синтаксис

```
Node previousNode();
```

Возвращаемое значение

Узел, предшествующий текущему узлу в исходном тексте документа и не отвергнутый фильтрами, или `null`, если такой узел отсутствует.

Описание

Этот метод устанавливает `currentNode` на предыдущий узел (в порядке исходного текста документа), который не отвергнут фильтрами, и возвращает этот узел. Если такой узел отсутствует или если поиск предыдущего узла выводит TreeWalker из поддерева `root`, `currentNode` остается неизменным и метод возвращает `null`.

Обратите внимание, что этот метод «выпрямляет» древовидную структуру документа и возвращает узлы в том порядке, в котором они расположены в исходном тексте документа. Вызов `previousNode()` может привести к перемещению текущего узла вниз, вбок или вверх по дереву документа. Этот тип выпрямляющего обхода также выполняется с помощью `NodeIterator.previousNode()`.

TreeWalker.previousSibling() DOM Level 2 Traversal

возвращает предыдущий смежный узел, не отвергнутый фильтрами

Синтаксис

```
Node previousSibling();
```

Возвращаемое значение

Предыдущий смежный узел текущего узла, который не отвергнут фильтрами, или `null`, если такой узел отсутствует.

Описание

Этот метод устанавливает `currentNode` на предыдущий смежный узел текущего узла, который не отвергнут фильтрами, и возвращает этот смежный узел. Если такого узла нет, `currentNode` остается неизменным и возвращается `null`.

UIEvent DOM Level 2 Events

сведения о событиях пользовательского интерфейса **Event → UIEvent**

Подынтерфейсы

MouseEvent

Свойства

`readonly long detail`

Сведения о событии (число). Для событий click, mousedown и mouseup (см. «MouseEvent») это поле обозначает количество нажатий: 1 – одинарное нажатие, 2 – двойное нажатие, 3 – тройное нажатие и т. д. Для событий DOMActivate значение этого поля, равное 1, соответствует обычной активации, равное 2 – «гиперактивации», например двойному нажатию или комбинации клавиш Shift-Enter.

`readonly AbstractView view`

Окно («вид», представление), в котором было сгенерировано событие.

Методы

`initUIEvent()`

Инициализирует свойства только что созданного объекта UIEvent, включая свойства, унаследованные от интерфейса Event.

Описание

Интерфейс UIEvent – это подынтерфейс Event, который определяет тип объекта Event, передаваемого событиями типов DOMFocusIn, DOMFocusOut и DOMActivate. Эти типы событий не часто применяются в веб-броузерах, но что более важно – интерфейс UIEvent является родительским интерфейсом для MouseEvent.

См. также

Event, MouseEvent; главу 19

UIEvent.initUIEvent()

DOM Level 2 Events

инициализирует свойства объекта UIEvent

Синтаксис

```
void initUIEvent(String typeArg,
                 boolean canBubbleArg,
                 boolean cancelableArg,
                 AbstractView viewArg,
                  long detailArg);
```

Аргументы

typeArg

Тип события.

canBubbleArg

Будет ли событие всплывать.

cancelableArg

Может ли событие быть отменено с помощью `preventDefault()`.

viewArg

Окно, в котором произошло событие.

detailArg

Свойство `detail` для события.

Описание

Этот метод инициализирует свойства `view` и `detail` данного объекта UIEvent, а также свойства `type`, `bubbles` и `cancelable`, унаследованные от интерфейса Event. Этот метод может быть вызван исключительно для только что созданных объектов UIEvent до того, как они будут переданы методу `EventTarget.dispatchEvent()`.

ViewCSS

см. AbstractView

Часть VI. Указатель классов, свойств, методов и обработчиков событий

Эта часть книги представляет собой указатель всех классов, свойств, методов и обработчиков событий в JavaScript. Данный индекс призван облегчить поиск справочного материала по этим темам.

Указатель классов, свойств, методов и обработчиков событий

Данный указатель поможет узнать, в каком именно справочном разделе следует искать интересующую вас информацию. Обозначение «[Базовый]» соответствует «Справочнику по базовому JavaScript», «[Клиентский]» – «Справочнику по клиентскому JavaScript», а «[DOM]» – «Справочнику по W3C DOM».

Если вы забыли, в каком классе определен метод, свойство или обработчик события, откройте этот указатель.

A

abbr: HTMLTableCellElement[DOM]

ABORT: Event[Клиентский]

above: Layer[Клиентский]

abs: Math[Базовый]

AbstractView: [DOM]

accept: HTMLInputElement[DOM]

acceptCharset: HTMLFormElement[DOM]

accessKey: HTMLAnchorElement[DOM], HTMLInputElement[DOM], HTMLTextAreaElement[DOM]

acos: Math[Базовый]

action: Form[Клиентский], HTMLFormElement[DOM]

add: HTMLSelectElement[DOM]

addEventListener: EventTarget[DOM]

ADDITION: MutationEvent[DOM]

alert: Window[Клиентский]

align: HTMLInputElement[DOM], HTMLTableCaptionElement[DOM], HTMLTableCellElement[DOM], HTMLTableColElement[DOM], HTMLTableElement[DOM], HTMLTableRowElement[DOM], HTMLTableSectionElement[DOM]

aLink: HTMLBodyElement[DOM]

alinkColor: Document[Клиентский]

all: Document[Клиентский], HTMLElement[Клиентский]

alt: HTMLInputElement[DOM]

altKey: MouseEvent[DOM]

Anchor: [Клиентский]

anchors: Document[Клиентский], HTMLDocument[DOM]

appCodeName: Navigator[Клиентский]

appendChild: Node[DOM]

appendData: CharacterData[DOM]

appendMedium: MediaList[DOM]

Applet: [Клиентский]

applets: Document[Клиентский], HTMLDocument[DOM]

apply: Function[Базовый]

appName: Navigator[Клиентский]

appVersion: Navigator[Клиентский]

Area: [Клиентский]

Arguments: [Базовый]

arguments: [Базовый], Function[Базовый]

Array: [Базовый]

asin: Math[Базовый]
atan: Math[Базовый]
atan2: Math[Базовый]
Attr: [DOM]
attrChange: MutationEvent[DOM]
attributes: Node[DOM]
ATTRIBUTE_NODE: Node[DOM]
attrName: MutationEvent[DOM]
AT_TARGET: Event[DOM]
availHeight: Screen[Клиентский]
availLeft: Screen[Клиентский]
availTop: Screen[Клиентский]
availWidth: Screen[Клиентский]
axis: HTMLTableCellElement[DOM]
azimuth: CSS2Properties[DOM]

B

back: History[Клиентский], Window[Клиентский]
background: CSS2Properties[DOM], HTMLBodyElement[DOM], Layer[Клиентский]
backgroundAttachment: CSS2Properties[DOM]
backgroundColor: CSS2Properties[DOM]
backgroundImage: CSS2Properties[DOM]
backgroundPosition: CSS2Properties[DOM]
backgroundRepeat: CSS2Properties[DOM]
BAD_BOUNDARYPOINTS_ERR: RangeException[DOM]
below: Layer[Клиентский]
bgColor: Document[Клиентский], HTMLBodyElement[DOM], HTMLTableCellElement[DOM], HTMLTableElement[DOM], HTMLTableRowElement[DOM], Layer[Клиентский]
blue: RGBColor[DOM]
BLUR: Event[Клиентский]
blur: HTMLAnchorElement[DOM], HTMLInputElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM], Input[Клиентский], Window[Клиентский]
body: HTMLDocument[DOM]
Boolean: [Базовый]
border: CSS2Properties[DOM], HTMLTableElement[DOM], Image[Клиентский]
borderBottom: CSS2Properties[DOM]
borderBottomColor: CSS2Properties[DOM]
borderBottomStyle: CSS2Properties[DOM]
borderBottomWidth: CSS2Properties[DOM]
borderCollapse: CSS2Properties[DOM]
borderColor: CSS2Properties[DOM]
borderLeft: CSS2Properties[DOM]
borderLeftColor: CSS2Properties[DOM]
borderLeftStyle: CSS2Properties[DOM]
borderLeftWidth: CSS2Properties[DOM]
borderRight: CSS2Properties[DOM]
borderRightColor: CSS2Properties[DOM]
borderRightStyle: CSS2Properties[DOM]
borderRightWidth: CSS2Properties[DOM]
borderSpacing: CSS2Properties[DOM]
borderStyle: CSS2Properties[DOM]
borderTop: CSS2Properties[DOM]
borderTopColor: CSS2Properties[DOM]
borderTopStyle: CSS2Properties[DOM]
borderTopWidth: CSS2Properties[DOM]
borderWidth: CSS2Properties[DOM]
bottom: CSS2Properties[DOM], Rect[DOM]
bubbles: Event[DOM]
BUBBLING_PHASE: Event[DOM]
Button: [Клиентский]
button: MouseEvent[DOM]

C

call: Function[Базовый], JSObject[Клиентский]
callee: Arguments[Базовый]
caller: Function[Базовый]
cancelable: Event[DOM]
caption: HTMLTableElement[DOM]
captionSide: CSS2Properties[DOM]
captureEvents: Document[Клиентский], Layer[Клиентский], Window[Клиентский]
CAPTURING_PHASE: Event[DOM]
CDATASection: [DOM]

CDATA_SECTION_NODE: Node[DOM]
ceil: Math[Базовый]
cellIndex: HTMLTableCellElement[DOM]
cellPadding: HTMLTableElement[DOM]
cells: HTMLTableRowElement[DOM]
cellSpacing: HTMLTableElement[DOM]
ch: HTMLTableCellElement[DOM], HTMLTableColElement[DOM], HTMLTableRowElement[DOM], HTMLTableSectionElement[DOM]
CHANGE: Event[Клиентский]
CharacterData: [DOM]
charAt: String[Базовый]
charCodeAt: String[Базовый]
charset: HTMLAnchorElement[DOM]
CHARSET_RULE: CSSRule[DOM]
Checkbox: [Клиентский]
checked: Checkbox[Клиентский], HTMLInputElement[DOM], Input[Клиентский], Radio[Клиентский]
childNodes: Node[DOM]
children: HTMLElement[Клиентский]
chOff: HTMLTableCellElement[DOM], HTMLTableColElement[DOM], HTMLTableRowElement[DOM], HTMLTableSectionElement[DOM]
className: HTMLElement[Клиентский], HTMLElement[DOM]
clear: CSS2Properties[DOM], Document[Клиентский]
clearInterval: Window[Клиентский]
clearTimeout: Window[Клиентский]
CLICK: Event[Клиентский]
click: HTMLInputElement[DOM], Input[Клиентский]
clientX: MouseEvent[DOM]
clientY: MouseEvent[DOM]
clip: CSS2Properties[DOM]
clip.bottom: Layer[Клиентский]
clip.height: Layer[Клиентский]
clip.left: Layer[Клиентский]
clip.right: Layer[Клиентский]
clip.top: Layer[Клиентский]
clip.width: Layer[Клиентский]
cloneContents: Range[DOM]
cloneNode: Node[DOM]
cloneRange: Range[DOM]
close: Document[Клиентский], HTMLDocument[DOM], Window[Клиентский]
closed: Window[Клиентский]
collapse: Range[DOM]
collapsed: Range[DOM]
color: CSS2Properties[DOM]
colorDepth: Screen[Клиентский]
cols: HTMLTextAreaElement[DOM]
colSpan: HTMLTableCellElement[DOM]
Comment: [DOM]
COMMENT_NODE: Node[DOM]
commonAncestorContainer: Range[DOM]
compareBoundaryPoints: Range[DOM]
complete: Image[Клиентский]
concat: Array[Базовый], String[Базовый]
confirm: Window[Клиентский]
constructor: Object[Базовый]
contains: HTMLElement[Клиентский]
content: CSS2Properties[DOM]
cookie: Document[Клиентский], HTMLDocument[DOM]
cookieEnabled: Navigator[Клиентский]
coords: HTMLAnchorElement[DOM]
cos: Math[Базовый]
Counter: [DOM]
counterIncrement: CSS2Properties[DOM]
counterReset: CSS2Properties[DOM]
createAttribute: Document[DOM]
createAttributeNS: Document[DOM]
createCaption: HTMLTableElement[DOM]
createCDATASection: Document[DOM]
createComment: Document[DOM]
createCSSStyleSheet: DOMImplementation[DOM]
createDocument: DOMImplementation[DOM]

createDocumentFragment: Document[DOM]
createDocumentType: DOMImplementation[DOM]
createElement: Document[DOM]
createElementNS: Document[DOM]
createEntityReference: Document[DOM]
createEvent: Document[DOM]
createHTMLDocument: DOMImplementation[DOM]
createNodeIterator: Document[DOM]
createProcessingInstruction: Document[DOM]
createRange: Document[DOM]
createTextNode: Document[DOM]
createTFoot: HTMLTableElement[DOM]
createTHead: HTMLTableElement[DOM]
createTreeWalker: Document[DOM]
CSS2Properties: [DOM]
CSSCharsetRule: [DOM]
cssFloat: CSS2Properties[DOM]
CSSFontFaceRule: [DOM]
CSSImportRule: [DOM]
CSSMediaRule: [DOM]
CSSPageRule: [DOM]
CSSPrimitiveValue: [DOM]
CSSRule: [DOM]
CSSRuleList: [DOM]
cssRules: CSSMediaRule[DOM], CSSStyleSheet[DOM]
CSSStyleDeclaration: [DOM]
CSSStyleRule: [DOM]
CSSStyleSheet: [DOM]
cssText: CSSRule[DOM], CSSStyleDeclaration[DOM], CSSValue[DOM]
CSSUnknownRule: [DOM]
CSSValue: [DOM]
CSSValueList: [DOM]
cssValueType: CSSValue[DOM]
CSS_ATTR: CSSPrimitiveValue[DOM]
CSS_CM: CSSPrimitiveValue[DOM]
CSS_COUNTER: CSSPrimitiveValue[DOM]
CSS_CUSTOM: CSSValue[DOM]
CSS_DEG: CSSPrimitiveValue[DOM]
CSS_DIMENSION: CSSPrimitiveValue[DOM]
CSS_EMS: CSSPrimitiveValue[DOM]
CSS_EXS: CSSPrimitiveValue[DOM]
CSS_GRAD: CSSPrimitiveValue[DOM]
CSS_HZ: CSSPrimitiveValue[DOM]
CSS_IDENT: CSSPrimitiveValue[DOM]
CSS_IN: CSSPrimitiveValue[DOM]
CSS_INHERIT: CSSValue[DOM]
CSS_KHZ: CSSPrimitiveValue[DOM]
CSS_MM: CSSPrimitiveValue[DOM]
CSS_MS: CSSPrimitiveValue[DOM]
CSS_NUMBER: CSSPrimitiveValue[DOM]
CSS_PC: CSSPrimitiveValue[DOM]
CSS_PERCENTAGE: CSSPrimitiveValue[DOM]
CSS_PRIMITIVE_VALUE: CSSValue[DOM]
CSS_PT: CSSPrimitiveValue[DOM]
CSS_PX: CSSPrimitiveValue[DOM]
CSS_RAD: CSSPrimitiveValue[DOM]
CSS_RECT: CSSPrimitiveValue[DOM]
CSS_RGBCOLOR: CSSPrimitiveValue[DOM]
CSS_S: CSSPrimitiveValue[DOM]
CSS_STRING: CSSPrimitiveValue[DOM]
CSS_UNKNOWN: CSSPrimitiveValue[DOM]
CSS_URI: CSSPrimitiveValue[DOM]
CSS_VALUE_LIST: CSSValue[DOM]
ctrlKey: MouseEvent[DOM]
cue: CSS2Properties[DOM]
cueAfter: CSS2Properties[DOM]
cueBefore: CSS2Properties[DOM]
currentNode: TreeWalker[DOM]
currentTarget: Event[DOM]
cursor: CSS2Properties[DOM]

D

data: CharacterData[DOM], ProcessingInstruction[DOM]
Date: [Базовый]
DBLCLICK: Event[Клиентский]
decodeURI: [Базовый]

decodeURIComponent: [Базовый]
defaultChecked: Checkbox[Клиентский], HTMLInputElement[DOM], Input[Клиентский], Radio[Клиентский]
defaultSelected: HTMLOptionElement[DOM], Option[Клиентский]
defaultStatus: Window[Клиентский]
defaultValue: HTMLInputElement[DOM], HTMLTextAreaElement[DOM], Input[Клиентский]
defaultView: Document[DOM]
deleteCaption: HTMLTableElement[DOM]
deleteCell: HTMLTableRowElement[DOM]
deleteContents: Range[DOM]
deleteData: CharacterData[DOM]
deleteMedium: MediaList[DOM]
deleteRow: HTMLTableElement[DOM], HTMLTableSectionElement[DOM]
deleteRule: CSSMediaRule[DOM], CSSStyleSheet[DOM]
deleteTFoot: HTMLTableElement[DOM]
deleteTHead: HTMLTableElement[DOM]
description: MimeType[Клиентский], Plugin[Клиентский]
detach: NodeIterator[DOM], Range[DOM]
detail: UIEvent[DOM]
dir: HTMLElement[DOM]
direction: CSS2Properties[DOM]
disabled: HTMLInputElement[DOM], HTMLOptionElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM], StyleSheet[DOM]
dispatchEvent: EventTarget[DOM]
display: CSS2Properties[DOM]
doctype: Document[DOM]
Document: [Клиентский], [DOM]
document: AbstractView[DOM], HTMLElement[Клиентский], Layer[Клиентский], Window[Клиентский]
DocumentCSS: [DOM]
documentElement: Document[DOM]
DocumentEvent: [DOM]
DocumentFragment: [DOM]
DocumentRange: [DOM]
DocumentStyle: [DOM]
DocumentTraversal: [DOM]
DocumentType: [DOM]
DocumentView: [DOM]
DOCUMENT_FRAGMENT_NODE: Node[DOM]
DOCUMENT_NODE: Node[DOM]
DOCUMENT_TYPE_NODE: Node[DOM]
domain: Document[Клиентский], HTMLDocument[DOM]
DOMException: [DOM]
DOMImplementation: [DOM]
DOMImplementationCSS: [DOM]
DOMSTRING_SIZE_ERR: DOMException[DOM]
DRAGDROP: Event[Клиентский]

E

E: Math[Базовый]
Element: [Клиентский], [DOM]
ElementCSSInlineStyle: [DOM]
elementFromPoint: Document[Клиентский]
elements: Form[Клиентский], HTMLFormElement[DOM]
ELEMENT_NODE: Node[DOM]
elevation: CSS2Properties[DOM]
embeds: Document[Клиентский]
emptyCells: CSS2Properties[DOM]
enabledPlugin: MimeType[Клиентский]
encodeURI: [Базовый]
encodeURIComponent: [Базовый]
encoding: CSSCharsetRule[DOM], Form[Клиентский]
enctype: HTMLFormElement[DOM]
endContainer: Range[DOM]
endOffset: Range[DOM]
END_TO_END: Range[DOM]
END_TO_START: Range[DOM]
entities: DocumentType[DOM]
Entity: [DOM]
EntityReference: [DOM]

ENTITY_NODE: Node[DOM]
ENTITY_REFERENCE_NODE: Node[DOM]
Error: [Базовый]
ERROR: Event[Клиентский]
escape: [Базовый]
eval: [Базовый], JSObject[Клиентский]
EvalError: [Базовый]
Event: [Клиентский], [DOM]
EventException: [DOM]
EventListener: [DOM]
eventPhase: Event[DOM]
EventTarget: [DOM]
exec: RegExp[Базовый]
exp: Math[Базовый]
expandEntityReferences: NodeIterator[DOM], TreeWalker[DOM]
extractContents: Range[DOM]

F

fgColor: Document[Клиентский]
filename: Plugin[Клиентский]
FileUpload: [Клиентский]
filter: NodeIterator[DOM], TreeWalker[DOM]
FILTER_ACCEPT: NodeFilter[DOM]
FILTER_REJECT: NodeFilter[DOM]
FILTER_SKIP: NodeFilter[DOM]
firstChild: Node[DOM], TreeWalker[DOM]
floor: Math[Базовый]
FOCUS: Event[Клиентский]
focus: HTMLAnchorElement[DOM], HTMLInputElement[DOM], HTMLSelect-Element[DOM], HTMLTextAreaElement[DOM], Input[Клиентский], Window[Клиентский]
font: CSS2Properties[DOM]
fontFamily: CSS2Properties[DOM]
fontSize: CSS2Properties[DOM]
fontSizeAdjust: CSS2Properties[DOM]
fontStretch: CSS2Properties[DOM]
fontStyle: CSS2Properties[DOM]
fontVariant: CSS2Properties[DOM]
fontWeight: CSS2Properties[DOM]
FONT_FACE_RULE: CSSRule[DOM]
Form: [Клиентский]
form: HTMLInputElement[DOM], HTMLOptionElement[DOM], HTMLSelectElement[DOM], HTMLTextArea-Element[DOM], Input[Клиентский]
forms: Document[Клиентский], HTMLDocument[DOM]
forward: History[Клиентский], Window[Клиентский]
Frame: [Клиентский]
frame: HTMLTableElement[DOM]
frames: Window[Клиентский]
fromCharCode: String[Базовый]
Function: [Базовый]

G

getAttribute: Element[DOM], HTMLElement[Клиентский]
getAttributeNode: Element[DOM]
getAttributeNodeNS: Element[DOM]
getAttributeNS: Element[DOM]
getClass: [Клиентский]
getComputedStyle: AbstractView[DOM]
getCounterValue: CSSPrimitiveValue[DOM]
getDate: Date[Базовый]
getDay: Date[Базовый]
getElementById: Document[DOM], HTMLDocument[DOM]
getElementsByName: HTMLDocument[DOM]
getElementsByTagName: Document[DOM], Element[DOM]
getElementsByTagNameNS: Document[DOM], Element[DOM]
getFloatValue: CSSPrimitiveValue[DOM]
getFullYear: Date[Базовый]
getHours: Date[Базовый]
getMember: JSObject[Клиентский]
getMilliseconds: Date[Базовый]
getMinutes: Date[Базовый]
getMonth: Date[Базовый]
getNamedItem: NamedNodeMap[DOM]

getNamedItemNS: NamedNodeMap[DOM]
getOverrideStyle: Document[DOM]
getPropertyCSSValue: CSSStyleDeclaration[DOM]
getPropertyPriority: CSSStyleDeclaration[DOM]
getPropertyValue: CSSStyleDeclaration[DOM]
getRectValue: CSSPrimitiveValue[DOM]
getRGBColorValue: CSSPrimitiveValue[DOM]
getSeconds: Date[Базовый]
getSelection: Document[Клиентский]
getSlot: JSObject[Клиентский]
getStringValue: CSSPrimitiveValue[DOM]
getTime: Date[Базовый]
getTimezoneOffset: Date[Базовый]
getUTCDate: Date[Базовый]
getUTCDay: Date[Базовый]
getUTCFullYear: Date[Базовый]
getUTCHours: Date[Базовый]
getUTCMilliseconds: Date[Базовый]
getUTCMinutes: Date[Базовый]
getUTCMonth: Date[Базовый]
getUTCSeconds: Date[Базовый]
getWindow: JSObject[Клиентский]
getYear: Date[Базовый]
Global: [Базовый]
global: RegExp[Базовый]
go: History[Клиентский]
green: RGBColor[DOM]

H

handleEvent: Document[Клиентский], HTMLElement[Клиентский], Layer[Клиентский], Window[Клиентский]
hasAttribute: Element[DOM]
hasAttributeNS: Element[DOM]
hasAttributes: Node[DOM]
hasChildNodes: Node[DOM]
hasFeature: DOMImplementation[DOM]
hash: Link[Клиентский], Location[Клиентский]
hasOwnProperty: Object[Базовый]
headers: HTMLTableCellElement[DOM]
height: CSS2Properties[DOM], HTMLTableCellElement[DOM], Image[Клиентский], Screen[Клиентский]
Hidden: [Клиентский]
hidden: Layer[Клиентский]
HIERARCHY_REQUEST_ERR: DOMException[DOM]
History: [Клиентский]
history: Window[Клиентский]
home: Window[Клиентский]
host: Link[Клиентский], Location[Клиентский]
hostname: Link[Клиентский], Location[Клиентский]
href: CSSImportRule[DOM], HTMLAnchorElement[DOM], Link[Клиентский], Location[Клиентский], StyleSheet[DOM]
hreflang: HTMLAnchorElement[DOM]
hspace: Image[Клиентский]
HTMLAnchorElement: [DOM]
HTMLBodyElement: [DOM]
HTMLCollection: [DOM]
HTMLDocument: [DOM]
HTMLDOMImplementation: [DOM]
HTMLElement: [Клиентский], [DOM]
HTMLFormElement: [DOM]
HTMLInputElement: [DOM]
HTMLOptionElement: [DOM]
HTMLSelectElement: [DOM]
HTMLTableCaptionElement: [DOM]
HTMLTableCellElement: [DOM]
HTMLTableColElement: [DOM]
HTMLTableElement: [DOM]
HTMLTableRowElement: [DOM]
HTMLTableSectionElement: [DOM]
HTMLTextAreaElement: [DOM]

I

id: HTMLElement[Клиентский], HTMLElement[DOM]
identifier: Counter[DOM]
ignoreCase: RegExp[Базовый]
Image: [Клиентский]

images: Document[Клиентский], HTMLDocument[DOM]
implementation: Document[DOM]
importNode: Document[DOM]
IMPORT_RULE: CSSRule[DOM]
index: HTMLOptionElement[DOM], Option[Клиентский]
indexOf: String[Базовый]
INDEX_SIZE_ERR: DOMException[DOM]
Infinity: [Базовый]
initEvent: Event[DOM]
initMouseEvent: MouseEvent[DOM]
initMutationEvent: MutationEvent[DOM]
initUIEvent: UIEvent[DOM]
innerHTML: HTMLElement[Клиентский]
innerText: HTMLElement[Клиентский]
Input: [Клиентский]
insertAdjacentHTML: HTMLElement[Клиентский]
insertAdjacentText: HTMLElement[Клиентский]
insertBefore: Node[DOM]
insertCell: HTMLTableRowElement[DOM]
insertData: CharacterData[DOM]
insertNode: Range[DOM]
insertRow: HTMLTableElement[DOM], HTMLTableSectionElement[DOM]
insertRule: CSSMediaRule[DOM], CSSStyleSheet[DOM]
internalSubset: DocumentType[DOM]
INUSE_ATTRIBUTE_ERR: DOMException[DOM]
INVALID_ACCESS_ERR: DOMException[DOM]
INVALID_CHARACTER_ERR: DOMException[DOM]
INVALID_MODIFICATION_ERR: DOMException[DOM]
INVALID_NODE_TYPE_ERR: RangeException[DOM]
INVALID_STATE_ERR: DOMException[DOM]
isFinite: [Базовый]
isNaN: [Базовый]
isPrototypeOf: Object[Базовый]
isSupported: Node[DOM]
item: CSSRuleList[DOM], CSSStyleDeclaration[DOM], CSSValueList[DOM], HTMLCollection[DOM], MediaList[DOM], NamedNodeMap[DOM], NodeList[DOM], StyleSheetList[DOM]

J

JavaArray: [Клиентский]
JavaClass: [Клиентский]
javaEnabled: Navigator[Клиентский]
JavaObject: [Клиентский]
JavaPackage: [Клиентский]
join: Array[Базовый]
JSObject: [Клиентский]

K

KEYDOWN: Event[Клиентский]
KEYPRESS: Event[Клиентский]
KEYUP: Event[Клиентский]

L

label: HTMLOptionElement[DOM]
lang: HTMLElement[Клиентский], HTMLElement[DOM]
language: Navigator[Клиентский]
lastChild: Node[DOM], TreeWalker[DOM]
lastIndex: RegExp[Базовый]
lastIndexOf: String[Базовый]
lastModified: Document[Клиентский]
Layer: [Клиентский]
layers: Layer[Клиентский]
left: CSS2Properties[DOM], Layer[Клиентский], Rect[DOM]
length: Arguments[Базовый], Array[Базовый], CharacterData[DOM], CSSRuleList[DOM], CSSStyleDeclaration[DOM], CSSValueList[DOM], Form[Клиентский], Function[Базовый], History[Клиентский], HTMLCollection[DOM], HTMLFormElement[DOM], HTMLSelectElement[DOM], Input[Клиентский], JavaArray[Клиентский], MediaList[DOM], NamedNodeMap[DOM], NodeList[DOM], Plugin[Клиентский], Select[Клиентский],

String[Базовый], StyleSheetList[DOM], Window[Клиентский]
letterSpacing: CSS2Properties[DOM]
lineHeight: CSS2Properties[DOM]
Link: [Клиентский]
link: HTMLBodyElement[DOM]
linkColor: Document[Клиентский]
links: Document[Клиентский], HTMLDocument[DOM]
LinkStyle: [DOM]
listStyle: Counter[DOM], CSS2Properties[DOM]
listStyleImage: CSS2Properties[DOM]
listStylePosition: CSS2Properties[DOM]
listStyleType: CSS2Properties[DOM]
LN10: Math[Базовый]
LN2: Math[Базовый]
LOAD: Event[Клиентский]
load: Layer[Клиентский]
localeCompare: String[Базовый]
localName: Node[DOM]
Location: [Клиентский]
location: Document[Клиентский], Window[Клиентский]
log: Math[Базовый]
LOG10E: Math[Базовый]
LOG2E: Math[Базовый]
lowsrc: Image[Клиентский]

M

margin: CSS2Properties[DOM]
marginBottom: CSS2Properties[DOM]
marginLeft: CSS2Properties[DOM]
marginRight: CSS2Properties[DOM]
marginTop: CSS2Properties[DOM]
markerOffset: CSS2Properties[DOM]
marks: CSS2Properties[DOM]
match: String[Базовый]
Math: [Базовый], Window[Клиентский]
max: Math[Базовый]
MAX_VALUE: Number[Базовый]
maxHeight: CSS2Properties[DOM]
maxLength: HTMLInputElement[DOM]
maxWidth: CSS2Properties[DOM]
media: CSSImportRule[DOM], CSSMediaRule[DOM], StyleSheet[DOM]
MediaList: [DOM]
mediaText: MediaList[DOM]
MEDIA_RULE: CSSRule[DOM]
message: Error[Базовый]
metaKey: MouseEvent[DOM]
method: Form[Клиентский], HTMLFormElement[DOM]
MimeType: [Клиентский]
mimeTypes: Navigator[Клиентский]
min: Math[Базовый]
MIN_VALUE: Number[Базовый]
minHeight: CSS2Properties[DOM]
minWidth: CSS2Properties[DOM]
MODIFICATION: MutationEvent[DOM]
MOUSEDOWN: Event[Клиентский]
MouseEvent: [DOM]
MOUSEMOVE: Event[Клиентский]
MOUSEOUT: Event[Клиентский]
MOUSEOVER: Event[Клиентский]
MOUSEUP: Event[Клиентский]
MOVE: Event[Клиентский]
moveAbove: Layer[Клиентский]
moveBelow: Layer[Клиентский]
moveBy: Layer[Клиентский], Window[Клиентский]
moveTo: Layer[Клиентский], Window[Клиентский]
moveToAbsolute: Layer[Клиентский]
multiple: HTMLSelectElement[DOM]
MutationEvent: [DOM]

N

name: Anchor[Клиентский], Attr[DOM], DocumentType[DOM], Error[Базовый], Form[Клиентский], HTMLAnchorElement[DOM], HTMLFormElement[DOM], HTMLInputElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM],

Image[Клиентский], Input[Клиентский], Layer[Клиентский], Plugin[Клиентский], Window[Клиентский]

namedItem: HTMLCollection[DOM]

NamedNodeMap: [DOM]

namespaceURI: Node[DOM]

NAMESPACE_ERR: DOMException[DOM]

NaN: [Базовый], Number[Базовый]

navigate: Window[Клиентский]

Navigator: [Клиентский]

navigator: Window[Клиентский]

NEGATIVE_INFINITY: Number[Базовый]

newValue: MutationEvent[DOM]

nextNode: NodeIterator[DOM], TreeWalker[DOM]

nextSibling: Node[DOM], TreeWalker[DOM]

Node: [DOM]

NodeFilter: [DOM]

NodeIterator: [DOM]

NodeList: [DOM]

nodeName: Node[DOM]

nodeType: Node[DOM]

nodeValue: Node[DOM]

normalize: Node[DOM]

Notation: [DOM]

notationName: Entity[DOM]

notations: DocumentType[DOM]

NOTATION_NODE: Node[DOM]

NOT_FOUND_ERR: DOMException[DOM]

NOT_SUPPORTED_ERR: DOMException[DOM]

noWrap: HTMLTableCellElement[DOM]

NO_DATA_ALLOWED_ERR: DOMException[DOM]

NO_MODIFICATION_ALLOWED_ERR: DOMException[DOM]

Number: [Базовый]

O

Object: [Базовый]

offset: Layer[Клиентский]

offsetHeight: HTMLElement[Клиентский]

offsetLeft: HTMLElement[Клиентский]

offsetParent: HTMLElement[Клиентский]

offsetTop: HTMLElement[Клиентский]

offsetWidth: HTMLElement[Клиентский]

onabort: Image[Клиентский]

onblur: Input[Клиентский], Window[Клиентский]

onchange: FileUpload[Клиентский], Input[Клиентский], Select[Клиентский], Textarea[Клиентский], Text[Клиентский]

onclick: Button[Клиентский], Checkbox[Клиентский], HTMLElement[Клиентский], Input[Клиентский], Link[Клиентский], Radio[Клиентский], Reset[Клиентский], Submit[Клиентский]

ondblclick: HTMLElement[Клиентский]

onerror: Image[Клиентский], Window[Клиентский]

onfocus: Input[Клиентский], Window[Клиентский]

onhelp: HTMLElement[Клиентский]

onkeydown: HTMLElement[Клиентский]

onkeypress: HTMLElement[Клиентский]

onkeyup: HTMLElement[Клиентский]

onload: Image[Клиентский], Window[Клиентский]

onmousedown: HTMLElement[Клиентский]

onmousemove: HTMLElement[Клиентский]

onmouseout: HTMLElement[Клиентский], Link[Клиентский]

onmouseover: HTMLElement[Клиентский], Link[Клиентский]

onmouseup: HTMLElement[Клиентский]

onmove: Window[Клиентский]

onreset: Form[Клиентский]

onresize: Window[Клиентский]

onsubmit: Form[Клиентский]

onunload: Window[Клиентский]

open: Document[Клиентский], HTMLDocument[DOM], Window[Клиентский]

opener: Window[Клиентский]

Option: [Клиентский]

options: HTMLSelectElement[DOM], Input[Клиентский], Select[Клиентский]

orphans: CSS2Properties[DOM]
outerHTML: HTMLElement[Клиентский]
outerText: HTMLElement[Клиентский]
outline: CSS2Properties[DOM]
outlineColor: CSS2Properties[DOM]
outlineStyle: CSS2Properties[DOM]
outlineWidth: CSS2Properties[DOM]
overflow: CSS2Properties[DOM]
ownerDocument: Node[DOM]
ownerElement: Attr[DOM]
ownerNode: StyleSheet[DOM]
ownerRule: CSSStyleSheet[DOM]

P

padding: CSS2Properties[DOM]
paddingBottom: CSS2Properties[DOM]
paddingLeft: CSS2Properties[DOM]
paddingRight: CSS2Properties[DOM]
paddingTop: CSS2Properties[DOM]
page: CSS2Properties[DOM]
pageBreakAfter: CSS2Properties[DOM]
pageBreakBefore: CSS2Properties[DOM]
pageBreakInside: CSS2Properties[DOM]
pageX: Layer[Клиентский]
pageY: Layer[Клиентский]
PAGE_RULE: CSSRule[DOM]
parent: Window[Клиентский]
parentElement: HTMLElement[Клиентский]
parentLayer: Layer[Клиентский]
parentNode: Node[DOM], TreeWalker[DOM]
parentRule: CSSRule[DOM], CSSStyleDeclaration[DOM]
parentStyleSheet: CSSRule[DOM], StyleSheet[DOM]
parse: Date[Базовый]
parseFloat: [Базовый]
parseInt: [Базовый]
Password: [Клиентский]
pathname: Link[Клиентский], Location[Клиентский]
pause: CSS2Properties[DOM]
pauseAfter: CSS2Properties[DOM]
pauseBefore: CSS2Properties[DOM]
PI: Math[Базовый]
pitch: CSS2Properties[DOM]
pitchRange: CSS2Properties[DOM]
pixelDepth: Screen[Клиентский]
platform: Navigator[Клиентский]
playDuring: CSS2Properties[DOM]
Plugin: [Клиентский]
plugins: Document[Клиентский], Navigator[Клиентский]
plugins.refresh: Navigator[Клиентский]
pop: Array[Базовый]
port: Link[Клиентский], Location[Клиентский]
position: CSS2Properties[DOM]
POSITIVE_INFINITY: Number[Базовый]
pow: Math[Базовый]
prefix: Node[DOM]
preventDefault: Event[DOM]
previousNode: NodeIterator[DOM], TreeWalker[DOM]
previousSibling: Node[DOM], TreeWalker[DOM]
prevValue: MutationEvent[DOM]
primitiveType: CSSPrimitiveValue[DOM]
print: Window[Клиентский]
ProcessingInstruction: [DOM]
PROCESSING_INSTRUCTION_NODE: Node[DOM]
prompt: Window[Клиентский]
propertyIsEnumerable: Object[Базовый]
protocol: Link[Клиентский], Location[Клиентский]
prototype: Function[Базовый]
publicId: DocumentType[DOM], Entity[DOM], Notation[DOM]
push: Array[Базовый]

Q

quotes: CSS2Properties[DOM]

R

Radio: [Клиентский]
random: Math[Базовый]

Range: [DOM]
RangeError: [Базовый]
RangeException: [DOM]
readOnly: HTMLInputElement[DOM], HTMLTextAreaElement[DOM]
Rect: [DOM]
red: RGBColor[DOM]
ReferenceError: [Базовый]
referrer: Document[Клиентский], HTMLDocument[DOM]
RegExp: [Базовый]
rel: HTMLAnchorElement[DOM]
relatedNode: MutationEvent[DOM]
relatedTarget: MouseEvent[DOM]
releaseEvents: Document[Клиентский], Layer[Клиентский], Window[Клиентский]
reload: Location[Клиентский]
REMOVAL: MutationEvent[DOM]
remove: HTMLSelectElement[DOM]
removeAttribute: Element[DOM], HTMLElement[Клиентский]
removeAttributeNode: Element[DOM]
removeAttributeNS: Element[DOM]
removeChild: Node[DOM]
removeEventListener: EventTarget[DOM]
removeMember: JSObject[Клиентский]
removeNamedItem: NamedNodeMap[DOM]
removeNamedItemNS: NamedNodeMap[DOM]
removeProperty: CSSStyleDeclaration[DOM]
replace: Location[Клиентский], String[Базовый]
replaceChild: Node[DOM]
replaceData: CharacterData[DOM]
Reset: [Клиентский]
RESET: Event[Клиентский]
reset: Form[Клиентский], HTMLFormElement[DOM]
RESIZE: Event[Клиентский]
resizeBy: Layer[Клиентский], Window[Клиентский]
resizeTo: Layer[Клиентский], Window[Клиентский]
rev: HTMLAnchorElement[DOM]
reverse: Array[Базовый]
RGBColor: [DOM]
richness: CSS2Properties[DOM]
right: CSS2Properties[DOM], Rect[DOM]
root: NodeIterator[DOM], TreeWalker[DOM]
round: Math[Базовый]
routeEvent: Document[Клиентский], Layer[Клиентский], Window[Клиентский]
rowIndex: HTMLTableRowElement[DOM]
rows: HTMLTableElement[DOM], HTMLTableSectionElement[DOM], HTMLTextAreaElement[DOM]
rowSpan: HTMLTableCellElement[DOM]
rules: HTMLTableElement[DOM]

S

scope: HTMLTableCellElement[DOM]
Screen: [Клиентский]
screen: Window[Клиентский]
screenX: MouseEvent[DOM]
screenY: MouseEvent[DOM]
scroll: Window[Клиентский]
scrollBy: Window[Клиентский]
scrollIntoView: HTMLElement[Клиентский]
scrollTo: Window[Клиентский]
search: Link[Клиентский], Location[Клиентский], String[Базовый]
sectionRowIndex: HTMLTableRowElement[DOM]
Select: [Клиентский]
SELECT: Event[Клиентский]
select: HTMLInputElement[DOM], HTMLTextAreaElement[DOM], Input[Клиентский]
selected: HTMLOptionElement[DOM], Option[Клиентский]
selectedIndex: HTMLSelectElement[DOM], Input[Клиентский], Select[Клиентский]
selectNode: Range[DOM]
selectNodeContents: Range[DOM]

selectorText: CSSPageRule[DOM], CSSStyleRule[DOM]
self: Window[Клиентский]
separator: Counter[DOM]
setAttribute: Element[DOM], HTMLElement[Клиентский]
setAttributeNode: Element[DOM]
setAttributeNodeNS: Element[DOM]
setAttributeNS: Element[DOM]
setDate: Date[Базовый]
setEnd: Range[DOM]
setEndAfter: Range[DOM]
setEndBefore: Range[DOM]
setFloatValue: CSSPrimitiveValue[DOM]
setFullYear: Date[Базовый]
setHours: Date[Базовый]
setInterval: Window[Клиентский]
setMember: JSObject[Клиентский]
setMilliseconds: Date[Базовый]
setMinutes: Date[Базовый]
setMonth: Date[Базовый]
setNamedItem: NamedNodeMap[DOM]
setNamedItemNS: NamedNodeMap[DOM]
setProperty: CSSStyleDeclaration[DOM]
setSeconds: Date[Базовый]
setSlot: JSObject[Клиентский]
setStart: Range[DOM]
setStartAfter: Range[DOM]
setStartBefore: Range[DOM]
setStringValue: CSSPrimitiveValue[DOM]
setTime: Date[Базовый]
setTimeout: Window[Клиентский]
setUTCDate: Date[Базовый]
setUTCFullYear: Date[Базовый]
setUTCHours: Date[Базовый]
setUTCMilliseconds: Date[Базовый]
setUTCMinutes: Date[Базовый]
setUTCMonth: Date[Базовый]
setUTCSeconds: Date[Базовый]
setYear: Date[Базовый]
shape: HTMLAnchorElement[DOM]
sheet: LinkStyle[DOM]
shift: Array[Базовый]
shiftKey: MouseEvent[DOM]
SHOW_ALL: NodeFilter[DOM]
SHOW_ATTRIBUTE: NodeFilter[DOM]
SHOW_CDATA_SECTION: NodeFilter[DOM]
SHOW_COMMENT: NodeFilter[DOM]
SHOW_DOCUMENT: NodeFilter[DOM]
SHOW_DOCUMENT_FRAGMENT: NodeFilter[DOM]
SHOW_DOCUMENT_TYPE: NodeFilter[DOM]
SHOW_ELEMENT: NodeFilter[DOM]
SHOW_ENTITY: NodeFilter[DOM]
SHOW_ENTITY_REFERENCE: NodeFilter[DOM]
SHOW_NOTATION: NodeFilter[DOM]
SHOW_PROCESSING_INSTRUCTION: NodeFilter[DOM]
SHOW_TEXT: NodeFilter[DOM]
siblingAbove: Layer[Клиентский]
siblingBelow: Layer[Клиентский]
sin: Math[Базовый]
size: CSS2Properties[DOM], HTMLInputElement[DOM], HTMLSelectElement[DOM]
slice: Array[Базовый], String[Базовый]
sort: Array[Базовый]
source: RegExp[Базовый]
sourceIndex: HTMLElement[Клиентский]
span: HTMLTableColElement[DOM]
speak: CSS2Properties[DOM]
speakHeader: CSS2Properties[DOM]
speakNumeral: CSS2Properties[DOM]
speakPunctuation: CSS2Properties[DOM]
specified: Attr[DOM]
speechRate: CSS2Properties[DOM]
splice: Array[Базовый]
split: String[Базовый]
splitText: Text[DOM]
sqrt: Math[Базовый]
SQRT1_2: Math[Базовый]
SQRT2: Math[Базовый]

src: HTMLInputElement[DOM], Image[Клиентский], Layer[Клиентский]
startContainer: Range[DOM]
startOffset: Range[DOM]
START_TO_END: Range[DOM]
START_TO_START: Range[DOM]
status: Window[Клиентский]
stop: Window[Клиентский]
stopPropagation: Event[DOM]
stress: CSS2Properties[DOM]
String: [Базовый]
Style: [Клиентский]
style: CSSFontFaceRule[DOM], CSSPageRule[DOM],CSSStyleRule[DOM], HTMLElement[Клиентский], HTMLElement[DOM]
styleSheet: CSSImportRule[DOM]
StyleSheet: [DOM]
StyleSheetList: [DOM]
styleSheets: Document[DOM]
STYLE_RULE: CSSRule[DOM]
Submit: [Клиентский]
SUBMIT: Event[Клиентский]
submit: Form[Клиентский], HTMLFormElement[DOM]
substr: String[Базовый]
substring: String[Базовый]
substringData: CharacterData[DOM]
suffixes: MimeType[Клиентский]
summary: HTMLTableElement[DOM]
surroundContents: Range[DOM]
SyntaxError: [Базовый]
SYNTAX_ERR: DOMException[DOM]
systemId: DocumentType[DOM], Entity[DOM], Notation[DOM]
systemLanguage: Navigator[Клиентский]

T

tabIndex: HTMLAnchorElement[DOM], HTMLInputElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM]
tableLayout: CSS2Properties[DOM]
tagName: Element[DOM], HTMLElement[Клиентский]
tan: Math[Базовый]
target: Event[DOM], Form[Клиентский], HTMLAnchorElement[DOM], HTMLFormElement[DOM], Link[Клиентский], ProcessingInstruction[DOM]
tBodies: HTMLTableElement[DOM]
test: RegExp[Базовый]
Text: [Клиентский], [DOM]
text: Anchor[Клиентский], HTMLBodyElement[DOM], HTMLOptionElement[DOM], Link[Клиентский], Option[Клиентский]
textAlign: CSS2Properties[DOM]
Textarea: [Клиентский]
textDecoration: CSS2Properties[DOM]
textIndent: CSS2Properties[DOM]
textShadow: CSS2Properties[DOM]
textTransform: CSS2Properties[DOM]
TEXT_NODE: Node[DOM]
tFoot: HTMLTableElement[DOM]
tHead: HTMLTableElement[DOM]
timeStamp: Event[DOM]
title: Document[Клиентский], HTMLDocument[DOM], HTMLElement[Клиентский], HTMLElement[DOM], StyleSheet[DOM]
toDateString: Date[Базовый]
toExponential: Number[Базовый]
toFixed: Number[Базовый]
toGMTString: Date[Базовый]
toLocaleDateString: Date[Базовый]
toLocaleLowerCase: String[Базовый]
toLocaleString: Array[Базовый], Date[Базовый], Number[Базовый], Object[Базовый]
toLocaleTimeString: Date[Базовый]
toLocaleUpperCase: String[Базовый]
toLowerCase: String[Базовый]
top: CSS2Properties[DOM], Layer[Клиентский], Rect[DOM], Window[Клиентский]
toPrecision: Number[Базовый]
toString: Array[Базовый], Boolean[Базовый], Date[Базовый], Error[Базовый],

Function[Базовый], JSObject[Клиентский], Number[Базовый], Object[Базовый], Range[DOM], RegExp[Базовый], String[Базовый]

toTimeString: Date[Базовый]

toUpperCase: String[Базовый]

toUTCString: Date[Базовый]

TreeWalker: [DOM]

type: CSSRule[DOM], Event[DOM], HTMLAnchorElement[DOM], HTMLInputElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM], Input[Клиентский], MimeType[Клиентский], Select[Клиентский], StyleSheet[DOM]

TypeError: [Базовый]

U

UIEvent: [DOM]

undefined: [Базовый]

unescape: [Базовый]

unicodeBidi: CSS2Properties[DOM]

UNKNOWN_RULE: CSSRule[DOM]

UNLOAD: Event[Клиентский]

unshift: Array[Базовый]

URIError: [Базовый]

URL: [Клиентский], Document[Клиентский], HTMLDocument[DOM]

useMap: HTMLInputElement[DOM]

userAgent: Navigator[Клиентский]

userLanguage: Navigator[Клиентский]

UTC: Date[Базовый]

V

vAlign: HTMLTableCellElement[DOM], HTMLTableColElement[DOM], HTMLTableRowElement[DOM], HTMLTableSectionElement[DOM]

value: Attr[DOM], Button[Клиентский], Checkbox[Клиентский], FileUpload[Клиентский], Hidden[Клиентский], HTMLInputElement[DOM], HTMLOptionElement[DOM], HTMLSelectElement[DOM], HTMLTextAreaElement[DOM], Input[Клиентский], Option[Клиентский], Password[Клиентский], Radio[Клиентский], Reset[Клиентский], Submit[Клиентский], Text[Клиентский], Textarea[Клиентский]

valueOf: Boolean[Базовый], Date[Базовый], Number[Базовый], Object[Базовый], String[Базовый]

verticalAlign: CSS2Properties[DOM]

view: UIEvent[DOM]

ViewCSS: [DOM]

visibility: CSS2Properties[DOM], Layer[Клиентский]

vLink: HTMLBodyElement[DOM]

vlinkColor: Document[Клиентский]

voiceFamily: CSS2Properties[DOM]

volume: CSS2Properties[DOM]

vspace: Image[Клиентский]

W

whatToShow: NodeIterator[DOM], TreeWalker[DOM]

whiteSpace: CSS2Properties[DOM]

widows: CSS2Properties[DOM]

width: CSS2Properties[DOM], HTMLTableCellElement[DOM], HTMLTableColElement[DOM], HTMLTableElement[DOM], Image[Клиентский], Screen[Клиентский]

Window: [Клиентский]

window: Layer[Клиентский], Window[Клиентский]

wordSpacing: CSS2Properties[DOM]

write: Document[Клиентский], HTMLDocument[DOM]

writeln: Document[Клиентский], HTMLDocument[DOM]

WRONG_DOCUMENT_ERR: DOMException[DOM]

X

x: Layer[Клиентский]

Y

y: Layer[Клиентский]

Z

zIndex: CSS2Properties[DOM], Layer[Клиентский]

Алфавитный указатель

Специальные символы

- (дефис), задание диапазона символов в классе символов, 179
– (минус), 82
  – (вычитание), оператор, 81
  – – (декремент), оператор, 42, 83, 102
  –= (присваивание), оператор, 94
  –Infinity, минус бесконечность, 49
! (восклицательный знак)
  ! (логическое НЕ), оператор, 86, 91
  != (не равно), оператор, 86, 89
    Netscape 4, реализация JavaScript 1.2, 209
  !== (неидентичность), , 86
\ (обратная косая черта), 50, 181
  \n в регулярных выражениях, 183
  escape-последовательности, строковые литералы и регулярные выражения, 178, 189
$ (знак доллара)
  в идентификаторах, 43
  поиск в строке по регулярному выражению, 575
  поиск конца строки, 184, 185
  якорный символ (регулярные выражения), 177
% (по модулю), оператор, 382
& (амперсанд)
  & (поразрядное И), оператор, 92
  && (логическое И), оператор , 90
  &= (присваивание), оператор, 94
' (одинарные кавычки) в строках, 50
() (круглые скобки)
  в инструкции if, 103
  в регулярных выражениях, 182–183
  задание порядка вычисления, 80
  оператор вызова функции, 55, 56, 100, 117, 128
¶ (перевод строки), 50
; (точка с запятой), 35, 41
  JavaScript и CSS, 377
  автоматическая вставка в JavaScript, 114
  завершение цикла do loop, 110
  пустые инструкции, 124
  разделитель инструкций, 41, 101
* (звездочка)
  *= (присваивание), оператор, 94
  квантификатор в регулярных выражениях, 181
  умножения оператор, 80, 81
+ (плюс)
  + (унарный плюс), оператор, 82
  ++ (инкремент), оператор, 42, 82, 102
  += (присваивание), оператор, 94
    добавление текста к свойству innerHTML, 336
  квантификатор в регулярных выражениях, 181
  конкатенация строк, 52, 81, 89
  преобразование объектов в строковом или числовом контексте, 194
  сложения оператор, 80, 81
  типы данных операндов, 79
, (запятая), оператор, 98, 102
  объединение нескольких выражений в цикле, 112
. (точка), 99
  классы символов регулярных выражений, 179
  оператор, 56, 99, 141, 158
/ (прямая косая черта), 30
  / (деление), оператор, 81
  //, в комментариях, 30, 42
  /* и */, в комментариях, 42
  в регулярных выражениях, 61, 177
: (двоеточие), отделение меток, 113

= (знак равенства)
= (присваивание), оператор, 80, 93
== (равенство), оператор, 83
Netscape 4, реализация JavaScript 1.2, 85, 209
правила определения равенства, 84
сравнение строк, 89
=== (идентичность), оператор, 60, 83
выражения case, проверка на идентичность, 108
правила определения идентичных значений, 84
? (вопросительный знак), 95
в регулярных выражениях, 180
встраивание аргументов в URL, 248
квантификатор в регулярных выражениях, 180
условный оператор, 95
@-правила, 394
^ (символ вставки)
XOR (исключающее ИЛИ), оператор, 92
поиск начала строки, 185
элементы классов символов с отрицанием, 179
{ } (фигурные скобки)
в объектных литералах, 57
в регулярных выражениях, 180
разделение блоков инструкций, 102, 117
| (вертикальная черта)
| (поразрядное ИЛИ), оператор, 92
|| (логическое ИЛИ), оператор, 91
в регулярных выражениях, 181, 183
~ (тильда), поразрядный оператор НЕ, 92, 351
> (больше), оператор, преобразование объектного типа данных, 194
>= (больше или равно), оператор, преобразование объектного типа данных, 194

Числа

16-разрядные целые, 735
32-разрядные целые, 91
64-разрядные целые, 735

A

<a>, теги, 278, 593
href, атрибут, 278
HTMLAnchorElement и, 810
Link, объект, создание, 669
onmouseover и onmouseout, обработчики событий, 223
отображение графики внутри, 271
about:, спецификатор протокола, 451
above, свойство, 660
abs(), функция, 534
AbstractView, объект, 736, 738
absolute, атрибут, позиционирование элементов, 366
acos(), функция, 535
action, свойство, 617
ActionScript (язык сценариев), 21
activeElement, свойство, 602
accept, свойство, 825
acceptNode(), метод, 865
accessKey, свойство, 825, 842
action, атрибут, URL javascript, в качестве значения, 225
action, свойство, 286
ActiveX, управляющие элементы
и объект Applet, 594
отображение вложенных данных в Internet Explorer, 283
проблемы безопасности, 449
add(), метод, 831
addEventListener(), метод, 410, 808
передача ссылки на функцию непосредственно методу, 412
смешивание исходной объектной модели и DOM Level 2, 423
addEventListener(), метод
alert(), метод, 24, 233, 705
align, свойство, 825, 833
aLink, свойство, 812
alinkColor, свойство, 260, 599
Document, объект, 269
all[], мас сив, 345, 602, 605, 628
alt, свойство, 825
altKey, свойство, 846
IE Event, 426, 613
MouseEvent, объект, 419
Anchor, объект, 246, 281, 593
anchors, свойство, 815
anchors[], массив, 246, 260, 599
animateCSS(), функция, 383

appCodeName, свойство, 241, 677
Applet, объект, 594
  и атрибут name, 262
<applet>, тег, 282
applets[], мас сив, 260, 282, 455, 599, 815
apply(), метод, 139
  Function, объект, 524
appName, свойство, 241, 677
appVersion, свойство, 241, 677
appendChild(), метод, 316, 331, 858
appendData(), метод, 334, 740
appendMedium(), метод, 845
<area>, тег, 278
arguments[], массив, 480
arguments, идентификатор, 135
Arguments, объект, 135, 480
  callee, свойство, 136
  length, свойство, 136
Array, объект, 482, 492
Array(), функция-конструктор, 58
  вызов, 167
as, ключевое слово, 44
asin(), функция, 535
ASCII, кодировка символов, 39
atan(), функция, 535
atan2(), функция, 535
Attr, объект, 318, 738
attrChange, свойство, 848
attributes[], массив, 318, 855
attrName, свойство, 848
attachEvent(), метод, 428
availHeight, свойство, 689
availLeft, свойство, 689
availTop, свойство, 689
availWidth, свойство, 689

B

\b, 184
  символ «забой» в классах символов регулярных выражений, 180
\B, метасимвол, поиск неграницы слова 185
back(), метод
  History, объект, 250, 626
  Window, объект, 706
Back, кнопка, 626
background, свойство, 661, 812
beginDrag(), функция (пример), 419
  событийная модель Netscape 4, 433
below, свойство, 661
  Document, объект, 260, 269, 600
    обходные решения проблем несовместимости платформ, 439
  Layer, объект, 661
blue, свойство, 884
blur(), метод
  HTMLAnchorElement, объект, 811
  HTMLInputElement, объект, 319, 828
  HTMLSelectElement, объект, 832
  HTMLTextAreaElement, 843
  Input, объект, 645
  Window, объект, 233, 245, 706
<body>, тег, 226, 604
  атрибуты alink, link и vlink, 260
  установка цветовых свойств, 269
body, свойство, 815
Boolean(), вызов в качестве функции, 195
Boolean, объект, 493
border, атрибут, 271
border, свойство, 274, 638
bottom, атрибут, 366
bottom, свойство, 884
boolean, тип данных, 735
booleanValue(), метод, 472
break, инструкция, 41, 114
  в инструкции switch, 107
  точки с запятой, автоматическая вставка, 114
bubbles, свойство, 804
Button, объект, 25, 289, 594
  обработчики событий, 595
<button>, тег, 595
  создание объектов Reset, 688
  создание объектов Submit, 695
button, свойство, 846
  IE Event, объект, 426, 613
  MouseEvent, объект, 418

C

C/C++
  char, набор символов, 49
  double, тип данных, 46
  JavaScript и, 16
  версия интерпретатора JavaScript на языке C, 21
  комментарии, поддержка в JavaScript, 42

логический тип данных, сравнение с JavaScript, 54
call(), метод, 139
JSObject, объект, 458, 656
Function, объект, 526
callee, свойство, 136, 481
caller, свойство, 527
calculate(), функция, 29
cancelBubble, свойство (Event в IE), 427, 428, 613
cancelable, свойство, 804
caption, свойство, 834
captureEvents(), метод, 432, 706
catch (инструкция try/catch/finally), 120
case, метка (инструкция switch), 106
CDATASection, объект, 739, 769
ceil(), функция, 536
cells, свойство, 838
cellIndex, свойство, 833
char, тип данных, 49
Java, в JavaScript, 470
charAt(), метод, 569
charCodeAt(), метод, 569
CharacterData, объект, 334, 739
charAt(), метод, 53
Checkbox, объект, 25, 275, 289, 293, 296, 596
checked, свойство, 825
Checkbox, объект, 297, 596
Input, объект, 642
Radio, объект, 297, 685
childNodes, свойство, 316, 327, 855
children[], массив, 345, 602, 628
className, свойство, 318, 319, 628, 819
CLASSPATH, переменная, 460
clear(), метод, 605
clearInterval(), метод, 234, 238, 707
clearTimeout(), метод, 234, 238, 707
click(), метод, 646
HTMLInputElement, объект, 828
click, события, инициализация, 424
clientInformation, свойство, 240
clientX, clientY, свойства
IE Event, объект, 426, 613
MouseEvent, объект, 419, 847
clientInformation, свойство, 702
clip, атрибут, 372
cloneContents(), метод, 354, 874
cloneNode(), метод, 859
cloneRange(), метод, 874
close(), метод
Document, объект, 260, 605
HTMLDocument, объект, 816
Window, объект, 233, 245, 708
ограничения по безопасности, 450
создание новых документов, 264, 279
closed, свойство, 232, 245, 699
colorDepth, свойство, 243
cols, свойство, 842
collapse(), метод, 875
collapsed, свойство, 871
complete, свойство, 274, 638
Comment, объект, 317, 743
commonAncestorContainer, свойство, 871
compareBoundaryPoints(), метод, 875
concat(), метод
Array, объект, 172, 484
String, объект, 570
confirm(), метод, 233, 708
constructor, свойство, 160, 551
contains(), метод, 630
continue, инструкция
ошибка в цикле do/while, 110, 115
cookie, свойство, 600, 606
cos(), функция, 536
code, свойство, 783
colors, цвет гиперссылки, 260
colorDepth, свойство, 690
cookie, 26
domain, атрибут, 307, 308
expires, атрибут, 306
path, атрибут, 306, 308
secure, атрибут, 307, 308
видимость, 306
исходная спецификация (веб-сайт), 307
пример, 310
свойство, 260, 305, 815
разбор, 309
сохранение, 307
ограничение броузера и сервера, 308
удаление, 308
чтение, 309
cookieEnabled, свойство, 678
Counter, объект, 395, 744
createTFoot(), метод, 836
createTHead(), метод, 836

createTreeWalker(), метод, 350, 775
createAttribute(), метод, 768
createAttributeNS(), метод, 769
createCaption(), метод, 836
createComment(), метод, 770
createCDATASection(), метод, 769
createCSSStyleSheet(), метод, 393
createDocument(), метод, 785
createDocumentFragment(), метод, 770
createDocumentType(), метод, 786
createElement(), метод, 335, 770
createElementNS(), метод, 771
createEntityReference(), метод, 771
createEvent(), метод, 423, 772
createHTMLDocument(), метод, 787
createNodeIterator(), метод, 350, 773
createRange(), метод, 353, 774
createProcessingInstruction(), метод, 774
createTextNode(), метод, 333, 335, 775
creator, свойство, 436
CSS (Cascading Style Sheets), каскадные таблицы стилей, 356
DOM API, 390
CSSStyleDeclaration, объект, 391
DOMImplementation, объект, 393
обход таблиц стилей, 393
перекрывающие стили, 392
Internet Explorer 4, 386
Netscape 4 Layer API, 347
атрибуты стилей, 25, 357
и объект Style, 694
версии, 363
определение и использование (пример), 364
отображение и видимость элемента, частичная видимость, 371–372
позиционирование элементов, 365
position, атрибут, 365
z-index (порядок в стеке), атрибут, 370
задание положения и размера, 366
пример, 373
работа со стилями в JavaScript, 375
анимации DHTML, 382
динамические гистограммы (пример), 379
названия CSS-атрибутов в JavaScript, 376
связывание с документами, 362
стилевые правила, применение к документам, 361
эквиваленты стилевых свойств в объекте Layer, 387
CSS2Properties, объект, 375, 391, 744
имена свойств и атрибуты CSS, 376
работа со свойствами стилей, 377
CSSCharsetRule, объект, 745
CSSFontFaceRule, объект, 746
CSSImportRule, объект, 746
CSSMediaRule, объект, 746
CSSPageRule, объект, 748
CSSPrimitiveValue, объект, 395, 749, 754
CSSRule, объект, 393, 755
cssRules[], массив, 393, 746, 762
CSSRuleList, объект, 756
CSSStyleDeclaration, объект, 391, 394, 757, 761
CSSStyleRule, объект, 394, 761
CSSStyleSheet, объект, 393, 762, 763
cssText, свойство, 755, 757, 764
CSSUnknownRule, объект, 763
CSSValue, объект, 394, 764
cssValueType, свойство, 394, 764
CSSValueList, объект, 394, 765
ctrlKey, свойство, 847
IE Event, объект, 426, 613
MouseEvent, объект, 419
currentNode, свойство, 888
currentStyle, свойство (IE 5 и более поздние), 392
currentTarget, свойство, 412, 804
Event, объект, 417

D

\d (цифры ASCII), 179
\D (цифры ASCII), любой символ, не являющийся цифрой), 180
data, свойство, 740, 870
Date, объект, 27, 46, 60, 494, 515
+, оператор, 194
Netscape 2, ошибки, 436
копирование, передача параметров и сравнение по ссылке, 201
локализованные значения, 163
преобразование в базовый тип данных, 85
список методов, 495

decodeURI(), функция, 515
decodeURIComponent(), функция, 515
defaultCharset, свойство, 602
defaultChecked, свойство, Checkbox, объект, 597
defaultStatus, свойство, 232, 237
отображения сообщения-приветствия, 238
description, свойство
MimeType, объект, 676
Plugin, объект, 683
debug(), функция, 335
default:, метка (инструкция switch), 107
defaultChecked, свойство, 826
Input, объект, 642
Radio, объект, 685
defaultSelected, свойство, 680, 829
defaultStatus, свойство, 699, 709
defaultValue, свойство, 642, 826, 842
defaultView, свойство, 766
delete, оператор, 97, 102, 142
deleteTFoot(), метод, 837
deleteTHead(), метод, 837
deleteCaption(), метод, 836
deleteCell(), метод, 839
deleteContents(), метод, 354, 876
deleteData(), метод, 334, 741
deleteMedium(), метод, 845
deleteRow(), метод, 837, 841
deleteRule(), метод
CSSMediaRule, объект, 747
CSSStyleSheet, объект, 762
добавление/удаление правил из таблиц стилей, 393
detach(), метод
NodeIterator, объект, 867
Range, объект, 877
detachEvent(), метод, 428
detail, свойство, 418, 893
DHTML (Dynamic HTML), динамический HTML, 19, 356
анимация с использованием CSS и JavaScript, 382
атрибут position в CSS и, 365
использование в Netscape 4, 386
кросс-платформенный анимационный сценарий, 389
проверка наличия возможностей для эффектов, 438
dir, свойство, 318, 819
display, атрибут, 371
display_time_in_status_line(), функция (пример), 238
dispatchEvent(), метод, 809
EventTarget, объект, 425
disabled, свойство, 826, 829, 830, 842, 885
do/while, циклы, 110
инструкция continue, 115
doctype, свойство, 766
Document, объект, 246, 599, 611
Internet Explorer, свойства, 602
Netscape
методы, 603
свойства, 602
document, свойство
HTMLElement, объект, 628
Window, объект, 233
DOM (Document Object Model), объектная модель документа, 19, 215, 261, 314, 731
API для таблиц стилей CSS, 390
CSS2Properties, объект, 375
CSSStyleDeclaration, объект, 391
DOMImplementation, объект, 393
обход таблиц стилей, 393
перекрывающие стили, 392
DOMException, объект, 782
EventException, объект, 806
HTML API, 318
динамическое создание оглавления, 338
добавление содержимого в документы, 335
модификация документа, 330
обход документа, 326
поиск определенных элементов в документе, 328
объекты DocumentFragment, 337
соглашения об именах, 319
Range API, 352
начальная и конечная позиции области, 353
работа с областями, 354
Traversal API, 349
NodeIterator и TreeWalker, 349
фильтрация, 350
версии (уровни) стандарта W3C, 11
древовидная структура, 315
узлы Node, 316

интерфейсы, независимые
от языка, 324
исходная событийная модель
не поддерживается, 401
модель, определяемая в броузерах, 19
обработка событий (Level 2), 408
буксировка элементов (пример), 419
интерфейсы и детализирующие свойства, 416
модули и типы событий, 413
распространение событий, 408
регистрация функций-обработчиков, 410, 412
синтезирование событий, 423
смешивание с исходной событийной моделью, 422
поддержка броузером стандарта W3C, 19, 25
проверка на совместимость, 321
работа с XML-документами, 342
совместимость
изменение документа, 346
с IE 4, 345
с Netscape 4, 347
со стандартом, 320
тестовые наборы, 323
стандарт W3C, 12
цветовые свойства (Document), отмена, 269
стандартный API и клиентский JavaScript, 12
уровни и возможности, 320
Document, объект, 259, 318, 766
HTML, 214
domain, свойство, 452, 600, 606
write(), метод, 22
апплеты, 282
дата последнего изменения, 27
динамически создаваемые документы, 264
изображения
замена, реализация ToggleButton, 275
обработчики событий, 273
имена, 262
методы, 260
обработчики событий и, 263
открытие документов, не содержащих HTML, 268
перехват событий (Netscape 4), 432
свойства, 260
создание, 785
ссылки
и безопасность в JavaScript, 279
обработчики событий, 280
стандарты и, 261
управление внешним видом и содержимым, 24
формы, 270
цветовые свойства, 269
якорные элементы, 281
навигационное окно со списком всех якорных элементов, 281
document, свойство
Document, объект, 737
Layer, объект, 387, 661
Window, объект, 214, 259, 699
document.write(), метод (Document, объект), 35
documentElement, свойство, 317, 766
DocumentFragment, объект, 337, 780
DocumentType, объект, 781
создание, 786
domain, атрибут cookie, 307, 308
domain, свойство, 261, 452, 815
DOMException, объект, 782
DOMImplementation, объект, 784, 789
Double, тип (Java), в JavaScript, 470
double, тип данных (C, C++), 46
doubleValue(), метод, 471

E

e или E (экспоненциальная) нотация, 47
E, константа (Math, объект), 537
ECMA-262, стандарт языка JavaScript, 11
версии, 18
ECMAScript
Netscape 4, несовместимость, 209
v1, символы Unicode в строковых литералах, 50
v3
инструкция throw и класс Error, 120
литералы массивов, 59
правила формирования идентификаторов, 43
синтаксис объектных литералов, 57

список глобальных переменных и функций, 44
список зарезервированных слов, 44
выражения, работающие в качестве литералов массива или объекта, 42
наборы символов, 40
управляющие последовательности, 51
Element, объект, 789, 802
elementFromPoint(), метод, 603, 607
elements, свойство, 823
elements[], массив, 301, 617, 619
else if, инструкции, 105
else, инструкции, 104
embeds[], массив, 283, 600
<embed>, тег, 283
enabledPlugin, свойство, 676
encodeURI(), функция, 516
encodeURIComponent(), функция, 517
endContainer, свойство, 871
encoding, свойство, 286, 617, 745
endOffset, свойство, 872
Entity, объект, 802
EntityReference, объект, 803
entities, свойство, 781
Error, объект, 46, 120, 518, 520
escape(), функция, 307, 520
eval(), метод, 521
JSObject, объект, 458, 474, 657
метод, строковые значения и объекты String, 63
EvalError, объект, 522
event, атрибут (тег <script>, IE), 225
event, ключевое слово, 422
Event, объект, 417, 612, 615, 803, 806
currentTarget, свойство, 412
Internet Explorer, 425
свойства, 613
Netscape, 431
интерфейсы DOM, события, Event, 413
как глобальная переменная, 427
константы битовой маски, 612
ограничения по безопасности на установку свойств, 451
event, свойство, 702
Window, объект, 427
EventException, объект, 806
EventListener, объект, 412, 807
eventPhase, свойство, 417, 804
EventTarget, объект, 411, 807, 810
exceptions, RangeException, 883
exec(), метод, 190, 562
exp(), функция, 537
expandEntityReferences, свойство, 866, 888
expando, свойство, 602
expires, атрибут cookie, 306
установка, 307
extractContents(), метод, 354, 877

F

factorial(), метод, 121
false, ключевое слово, 91
fgColor, свойство, 260, 600
Document, объект, 269
FileUpload, объект, 615
filter, свойство, 866, 889
firstChild, свойство, 316, 327, 855
firstChild(), метод, 891
<fieldset> и <label>, теги, 300
filename, свойство, 683
FileUpload, объект, 289, 298
ограничения по безопасности, 450
finally, блок (инструкция try/catch/finally), 120
fixed, позиционирование элементов, 366
Flash, язык ActionScript, 21
float, ключевое слово, 377
float, тип данных, 735
floor(), функция, 537
focus(), метод, Window, объект, 233, 245
for, атрибут (тег <script>, IE), 225
for, циклы, 110
инструкция continue, 115
с метками, 115
for/in, циклы, 36, 112
continue, инструкция, 115
var, инструкция, 66
использование с ассоциативными массивами, 159
перечисляемые свойства объекта, 142, 551, 554
список всех глобальных переменных, 531
<form>, тег, 286, 618
action, атрибут, javascript: URL , в качестве значения, 225

name, атрибут, 293
target, атрибут, 622
Form, объект, 25, 215, 270, 286, 617, 622
и атрибут name, 262
form, свойство, 826, 829, 830, 842
Input, объект, 642
элементы форм, 294
forms[], мас сив, 215, 261, 262, 270, 318, 600, 815
forward(), метод
History, объект, 250, 626
Window, объект, 710
Forward, кнопка, 626
focus(), метод
HTMLAnchorElement, объект, 811
HTMLInputElement, объект, 319, 828
HTMLSelectElement, объект, 832
HTMLTextAreaElement, 843
Input, объект, 646
Window, объект, 710
frames[], массив, 233, 252, 699
<frameset> и <frame>, теги, 253
Frame, объект, 622
fromCharCode(), метод, 570
fromElement, свойство, 426, 613
function, ключевое слово, 256
function, инструкция, 117, 127
и конструктор Function(), 130

G

g, атрибут (глобальный поиск), 561, 564
g, флаг (регулярные выражения), 185, 191
getAttribute(), метод, 318, 346, 792
HTMLELement, объект, 631
getAttributeNode(), метод, 739, 793
getAttributeNodeNS(), метод, 793
getAttributeNS(), метод, 794
getClass, функция, 464, 623
getComputedStyle(), метод, 391, 737
getCounterValue(), метод, 751
getDate(), метод, 499
getDay(), метод, 499
getElementById(), метод, 776
getElementsByName(), метод, 816
getElementsByTagName(), метод
Document, объект, 329, 777, 778
Element, объект, 794, 795
getFloatValue(), метод, 395, 752
getFullYear(), метод, 499
getHours(), метод, 499
getMember(), метод, 458, 657
getMilliseconds(), метод, 500
getMinutes(), метод, 500
getMonth(), метод, 500
getNamedItem(), метод, 851
getNamedItemNS(), метод, 851
getOverrideStyle(), метод, 392, 778
getPropertyCSSValue(), метод, 394, 758
getPropertyPriority(), метод, 759
getPropertyValue(), метод, 391, 394
getRectValue(), метод, 752
getRGBColorValue(), метод, 395, 753
getSeconds(), метод, 500
getSelection(), метод, 607
getSlot(), метод, 458, 658
getStringValue(), метод, 753
getTime(), метод, 500
getTimezoneOffset(), метод, 501
getUTCDate(), метод, 501
getUTCDay(), метод, 501
getUTCFullYear(), метод, 502
getUTCHours(), метод, 502
getUTCMilliseconds(), метод, 502
getUTCMinutes(), метод, 502
getUTCMonth(), метод, 502
getUTCSeconds(), метод, 503
getWindow(), метод, 457, 658
getYear(), метод, 503
global, свойство, 564
регулярные выражения, 191
GMT (Greenwich Mean Time), время по Гринвичу, 495, 498
локальное время, отличие от, 501
строковое представление даты и времени, 511
go(), метод, 627
ошибки, 251
green, свойство, 885
GUI (Graphical User Interfaces), графические пользовательские интерфейсы, 217
HTML-формы, содержащие элементы графического интерфейса, 270

H

handleEvent(), метод, 413, 711
hasAttribute(), метод, 796
hasAttributeNS(), метод, 796

hasAttributes(), метод, 859
hasChildNodes(), метод, 860
hasFeature(), метод, 321, 787
hasOwnProperty(), метод, 164, 551, 552
hash, свойство
  Link, объект, 668
  Location, объект, 673
<head>, тег, присутствие сценариев, 226
height(), метод, 368
height, свойство
  Document, объект, 602, 612
  Image, объект, 274, 639
  Screen, объект, 243, 690
Hidden, объект, 25, 289, 300, 624
hidden, свойство, 661
History, объект, 25, 250, 625, 627
  ограничения по безопасности, 449
history, свойство, 233, 250, 699
host, свойство
  Link, объект, 668
  Location, объект, 247, 674
hostname, свойство
  Link, объект, 668
  Location, объект, 674
hours, UTC time, 502
home(), метод, 711
href, атрибут, 278
href, свойство, 746, 885
  Link, объект, 280, 668
  Location, объект, 248, 674
hspace, свойство, 274, 639
HTML (Hypertext Markup Language), язык разметки гипертекста
  DOM API, 318
    соглашения об именах, 319
  Dynamic, 19
  src, атрибут, 639
  встраивание JavaScript в, 22, 217, 222
    <script>, теги, 218
    URL, 224
  динамически создаваемые документы, 264
  древовидная структура документа, 315
  кавычки в строках, 50
  комментарии, 445
  процесс анализа, 226
  ссылки, 278
  стандарт W3C DOM, 262
  таблицы стилей CSS, связь с документами, 362
HTMLAnchorElement, объект, 810
HTMLBodyElement, объект, 812
HTMLCollection, объект, 813
HTMLDocument, объект, 262, 318, 815
  создание, 787
HTMLElement, объект, 318, 627, 638, 818, 823
  Internet Explorer 4, 346
HTMLEvents, модуль, 413, 417
HTMLFormElement, объект, 319, 823
HTMLInputElement, объект, 319, 825
HTMLOptGroupElement, объекты, 300
HTMLOptionElement object, 829
HTMLSelectElement, объект, 830
HTMLTableCaptionElement, объекты, 833
HTMLTableCellElement, объект, 833
HTMLTableColElement, объект, 834
HTMLTableElement, объект, 834, 838
HTMLTableRowElement, объект, 838
HTMLTableSectionElement, объект, 840
HTMLTextAreaElement, объект, 842
HTML-атрибуты
  class, 319, 362
  href, 278
  id, 348
    и правила таблиц стилей, 362
  language, 219, 441
  lowsrc, 275
  multiple, 299
  name, 281
  onblur, 713
  onclick, 24, 223
  onunload, 717
  onfocus, 715
  onload, 715
  onmove, 716
  onresize, 717
  src, 222, 227
  style, 361
  target, 254, 622
  type, 219
  значение, 688
  обработчики событий, 397, 400
    onmousedown, 422
HTML-теги
  <a>, 278, 281
  <body>, и значения свойств Document, 604

<button>
создание объектов Reset, 688
создание объектов Submit, 695
<embed>, 600
<form>, 286, 618
<frame>, 253
<img>, 27, 274 640
<input>, 595, 597
создание объекта Submit, 694
создание элемента Hidden, 624
создание элемента Password, 683
создание элемента Radio, 686
создание элемента Reset, 688
создание элемента Text, 696
links[], 278
<noscript>, 446
<option>, 681
<select>, 691
<textarea>, 698
не поддерживающие атрибуты обработчиков событий, 399
регистр, 329
соответствующие HTMLElement, 820
строки, 567
HTML-формы, 270
forms[], массив, 259
взаимодействие с JavaScript, 25, 29
проверка входных данных, 26
обработчики событий, 23
HTTP-запросы
user-agent, заголовок, 678
определение броузера по полю User-Agent, 440
HTTP-соединения и cookie, 307

I

i, атрибут, поиск
не чувствительный к регистру, 564
чувствительный к регистру, 561
i, флаг (регулярные выражения), 185, 191
id, атрибут
Layer, объект, 348
и правила таблиц стилей, 362
id, свойство, 318, 628, 819
identifier, свойство, 744
if, инструкция, 95, 103
if/else, инструкции, 103
и инструкция else if, 105
логические значения, 53
<iframe>, тег, 253
ignoreCase, свойство, 191, 564
Image, объект, 638, 640
name, атрибут и, 262
внеэкранные изображения экрана и кэширование, 272
замена изображений
src, свойство, 271
реализация ToggleButton, 275
обработчики событий, 273
свойства, размер, границы и разрешающая способность, 274
images[], массив, 261, 271, 600, 815
<img>, тег, 274, 600, 640
implementation, свойство, 321, 766
importNode(), метод, 779
in, оператор, 88
in, инструкция (циклы for/in), 112
–Infinity, минус бесконечность, 546
Infinity, свойство, 528
init(), метод апплета, и JavaScript, 459
initEvent(), метод, 805
initMouseEvent(), метод, 847
initMutationEvent(), метод, 850
initUIEvent(), метод, 894
index, свойство, 829
Array, массив, 188
Option, объект, 681
indexOf(), метод, 52, 309
innerHTML, свойство, 336, 347, 628
innerHeight, свойство, 701
innerText, свойство, 347
Anchor, объект, 281
HTMLElement, объект, 628
innerWidth, свойство, 701
Input, объект, 642, 652
input, свойство, 188
<input>, теги
обработчик события onchange, 223
создание кнопок, 595
создание объекта Submit, 694
создание элемента Hidden, 624
создание элемента Password, 683
создание элемента Reset, 688
создание элементов Checkbox, 597
создание элементов Text, 696
insertAdjacentHTML(), метод, 631
insertAdjacentText(), метод, 631
instanceof, оператор, 88
insertBefore(), метод, 316, 860
insertCell(), метод, 839

insertData(), метод, 334, 741
insertNode(), метод, 878
insertRow(), метод
HTMLTableElement, 837
HTMLTableSectionElement, 841
insertRule(), метод
CSSMediaRule, объект, 747
CSSStyleSheet, объект, 393, 763
insertAdjacentHTML(), метод, 347
insertAdjacentText(), метод, 347
internalSubset, свойство, 781
Integer, тип (Java), в JavaScript, 470
Internet Explorer, 11, 19
elementFromPoint(), метод (Document), 603, 607
navigate(), метод (Window), 704, 712
scrollLeft и scrollTop, свойства (docu ment.body), 419
версия 3
cookie, свойство объекта Document, 309
ошибки go(), 251
версия 4
DHTML, 386
innerText, свойство тега a, 281
userLanguage, свойство, 678
совместимость DOM, 345
версия 5
currentStyle, свойство, 392
соответствие стандарту DOM, 323
версия 6
CSS-атрибуты положения и размера, 369
совместимость со стандартом DOM, 321
возможности клиентского JavaScript в различных версиях броузера, 19
встроенный VBScript, 20
зоны безопасности, 452
использование интерпретатора JScript, 21
нахождение определения класса JSObject, 460
нестандартный JavaScript, 225
объекты Java, рассмотрение как управляющих элементов ActiveX, 454
отображение вложенных данных с помощью ActiveX, 283
поддержка Java, 282
поддержка объекта Navigator, 240
свойства объекта Document, 602
свойства объекта Event, 613
свойства объекта Window, 702
свойства изображения, 275
событийная модель, 397, 425
Event, объект, 425
всплывание события, 427
перетаскивание элементов документа (пример), 428
регистрация обработчиков событий, 428
is, ключевое слово, 44
isNaN(), функция, 532
ISO Latin-1, кодировка символов, 39
isPrototypeOf(), метод, 165, 551, 553
isNaN(), функция, 49, 84
isSupported(), метод, 861
item(), метод
CSSRuleList, объект, 756
CSSStyleDeclaration, объект, 759
CSSValueList, 765
HTMLCollection, объект, 814
MediaList, объект, 846
NamedNodeMap, объект, 851
NodeList, объект, 869
StyleSheetList, объект, 886

J

Java, 16, 17
char, набор символов, 49
апплеты, 27, 282
интерпретатор JavaScript (Rhino), 21
использование классов непосредственно из JavaScript, 461
массивы, 466
методы, использование в JavaScript, 467
объекты, 465
пакеты, 462
применение JavaScript из, JSObject, класс, 457
тип данных, 464
манипуляции с типами JavaScript, 473
управление из JavaScript, 455
java, свойство, 701
JavaClass, объект, 464, 653

JavaScript
delete, оператор, 97
Java, применение из, JSObject, класс, 457
name, свойство (Window), 1.0 и 1.1, 255
switch, инструкция, отличия от других языков, 108
безопасность, 448
зоны безопасности и подписанные сценарии, 452
ограничения, 450
стратегия общего происхождения, 451
версии, 18
1.1, счетчик ссылок для сборки мусора, 205
1.2
ошибка в инструкции continue, 115
указание в атрибуте language, 220
совместимость, 441
явная проверка, 442
заблуждения, 17
зарезервированные слова, 43
кавычки в строках, 50
как универсальный язык программирования, 21, 24
классы Java, непосредственное использование, 461, 464
JavaArray, класс, 466
JavaObject, класс, 465
методы Java, 467
клиентский и базовый, 20
лексическая структура, 39
на стороне сервера, 440
JavaPackage, класс, 462
совместимость с броузерами, не поддерживающими JavaScript, 445
типы данных, преобразование Java-типов в JavaScript, 467
типы, необязательность указания, 16
управление Java, 455
управляющие последовательности, 51
javascript:, спецификатор псевдопротокола, 35, 224, 230
включение JavaScript на стороне клиента, 224
JavaArray, объект, 652
javaEnabled(), метод, 679
JavaObject, объект, 465, 654
передача из JavaScript в Java, 469
преобразование JavaScript, 471
создание новых, 465
JavaPackage, объект, 462, 655
join(), метод, 170, 485
JScript, интерпретатор, 21
JSObject, объект, 457, 473, 656, 660
применение апплетов, 459
js-файлы, встраивание Ja vaScript в теги <script>, 222

K

keyCode, свойство, Event, объект (Internet Explorer), 426, 613

L

label, свойство, 829
lang, свойство, 318, 628, 819
language, атрибут, 85, 219, 441
указание версии JavaScript, 220
language, свойство, 678
lastChild(), метод, 891
lastChild, свойство, 327, 855
Node, объект, 316
lastIndex, свойство, 191, 564
и методы объекта String, 190
lastIndexOf(), метод, 572
lastModified, свойство, дата изменения документа, 261
Latin-1, набор символов, управляющие последовательности, 51
<layer>, тег, 348, 388, 663
Layer API, Netscape, 386
Layer, объект, 386
свойства объекта Layer, имеющие CSS-эквиваленты, 387
Layer, объект, 660, 668
перехват событий (Netscape 4), 432
смежные объекты, расположение в стеке, 662
layers[], массив, 602, 661
layerX, layerY, свойства (Document), 612
left, атрибут, 366, 368
left, свойство, 387, 884
Layer, объект, 662
length, свойство, 740, 756, 757, 765, 813, 824, 830, 844, 850, 869, 886

Arguments, объект, 136, 481, 482
Array, объект, 483, 485
Form, объект, 617
Function, объект, 137, 527
History, объект, 250, 625
Input, объект, 643
JavaArray, объект, 466
Plugin, объект, 683
Select, объект, 690
String, объект, 572
Window, объект, 699
массивы, 167, 169
строки, 52
Link, объект, 278, 668, 673
обработчики событий, 280
link, свойство, 812
linkColor, свойство, 260, 601
Document, объект, 269
links[], массив, 261, 278, 601, 608, 815
Document, объект , цвет гиперссылки, 260
LinkStyle, объект, 844
listStyle, свойство, 744
LiveConnect, 454
типы данных, 462
JavaArray, класс, 466
JavaObject, класс, 465
и объект, 594
работа с типами Java из JavaScript, 467
работа с типами JavaScript в Java, 473
LiveVideo, подключаемый модуль (Netscape), 284
ln10, константа, 538
ln2, константа, 538
localeCompare(), метод, 573
load(), метод
Document, объект, 343
Layer, объект, 664
loadXML(), функция, 343
localeCompare(), метод, 87
localName, свойство, 855
Location, объект, 25, 215, 247, 673, 676
свойства, 673
навигационная панель, 251
location, свойство, 215
Document, объект, 261, 601
HTMLDocument, объект, 318
Window, объект, 233, 247, 249, 700
log(), функция, 538
log10e, константа, 539
log2e, константа, 539
long, тип данных, 735
lowsrc, атрибут, 275, 639
l-значения, 79

M

m, атрибут (многострочный поиск), 561
m, флаг (регулярные выражения), 185, 191
Macromedia Flash, программа, 21
magic cookie, 305
mailto, 450
maketoc(), функция, 338
margin, свойство, установка, 378
match(), метод, 188, 573
и свойство lastIndex, 191
Math, объект, 48, 542
max(), функция, 539
Math, свойство, 700
MAX_VALUE, константа, 544
maxLength, свойство, 826
media, свойство, 746, 747, 885
MediaList, объект, 844
mediaText, свойство, 844
method, свойство, 286, 617
metaKey, свойство, 847
MouseEvent, объект, 419
message, свойство
Error, объект, 61, 518, 519
EvalError, объект, 522
RangeError, объект, 560
ReferenceError, объект, 561
SyntaxError, объект, 583
TypeError, объект, 584
URIError, объект, 586
Microsoft
JScript, интерпретатор, 21
VBScript
атрибут language, 219
встраивание в броузеры, 20
интерпретатор JScript, 11
версии, 18
min(), функция, 539
MIN_VALUE, константа, 545
MimeType, объект, 676
mimeTypes[], массив, 678
MIME-типы
документы, не содержащие HTML, 268
задание языка сценариев, 219

modifiers, свойство, 612
mousedown, обработчик события,
вызов из функции beginDrag(), 419
события, инициализация, 424
MouseEvent, объект, 418, 846
интерфейсы DOM, события,
MouseEvent, 414
moveAbove(), метод, 388, 665
moveBelow(), метод, 388, 665
moveTo(), метод
Layer, объект, 388, 666
Window, объект, 234, 245, 711
moveBy(), метод
Layer, объект, 388, 666
Window, объект, 234, 245, 711
moveToAbsolute(), метод, 666
Mozilla, броузер, 11
CSS-атрибуты положения
и размера, 370
готовый код клиентских модулей
проверки окружения, 439
интерпретатор JavaScript с открытым исходным кодом, веб-сайт, 21
совместимость со стандартом DOM,
321
тестовые наборы, 323
multiple, атрибут (<select>), 299
multiple, свойство, 830
multiline, свойство (RegExp), 191
MutationEvent, объект, 419, 848
интерфейсы DOM, события, 414

N

name, атрибут (<frame>), 253
name, свойство
Anchor, объект, 593
Error, объект, 518, 519
EvalError, объект, 523
Form, объект, 618
RangeError, объект, 560
ReferenceError, объект, 561
SyntaxError, свойство, 583
TypeError, объект, 584
URIError, объект, 586
Window, объект, 233, 255
NaN (нечисло), 49, 198, 528, 532, 542,
545
isNaN(), функция, 532
преобразование строк в числа, 198
результат операции 0/0, 81
сравнение значений, 88, 84
name, атрибут, 293, 348
<img>, тег, 271
объекты Form, Image и Applet, 262
элементы, 293
name, свойство, 738, 781, 826, 830, 842
Anchor, объект, 281
Image, объект, 639
Input, объект, 643, 647
Layer, объект, 662
Plugin, объект, 684
Window, объект, 700, 712
NamedNodeMap, объект, 850
namespace, ключевое слово, 44
namespaceURI, свойство, 855
namedItem(), метод, 814
navigate(), метод, 712
Navigator, объект, 27, 240, 677, 680
navigator, свойство, 240, 439, 700
NEGATIVE_INFINITY, константа, 545
Netscape
Document, объект, методы, 603
Document, объект, свойства, 602
Event, объект, константы, 612
Event, объект, свойства, 612
Java, поддержка, 282
JavaScript 1.2 (атрибут language тега <script>)
и методы Array, 174
JavaScript
версии, 18
встроенный в веб-серверы, 17
интерпретатор, 11
LiveConnect, 454
непосредственное использование Java-классов, 461
Window, объект, методы, 703
Window, объект, свойства, 701
версия 2
bgColor, свойство, обходное решение на платформе Unix, 439
ошибки go(), 251
ошибки в объекте Date, 436
теги <script>, 222
версия 3
отсутствие поддержки обработчика события onclick, 225
преобразование данных LiveConnect, 470
сборка мусора (JavaScript 1.1),
204

версия 4
DHTML, 386
JavaScript, нестандартное применение, 225
text, свойство тега <a>, 281
безопасность, подписанные сценарии, 453
несовместимость JavaScript 1.2, 220
операторы равенства и неравенства, 85
событийная модель, 397, 431
совместимость DOM, 347
версия 6
CSS-атрибуты положения и размера, 370
базирующийся на коде Mozilla, 11
безопасность, подписанные сценарии, 453
обработчик onerror, ошибки, 239
совместимость со стандартом DOM, 321
возможности клиентского JavaScript в различных версиях броузера, 19
модули подключаемые, отображение вложенных данных, 283
свойства изображения, 275
тестовые наборы для совместимости со стандартом DOM, 323
чувствительность к регистру, 40
netscape, свойство, 701
new, оператор, 60, 63, 96, 143
JavaClass, объект и, 465
использование с функцией-конструктором, 143
создание объектов, 140
news, 450
newValue, свойство, 848
nextNode(), метод
NodeIterator, объект, 349, 868
TreeWalker, объект, 891
nextSibling(), метод, 892
nextSibling, свойство, 327, 855
Node, объект, 316
normalize(), метод, 862
<noscript>, тег, 446
Node, объект, 854, 864
свойства, 327
NodeFilter, объект, 350, 864
NodeIterator, объект, 866, 868
создание, 773
фильрация, 350
NodeList, объект, 329, 331, 869
nodeName, свойство, 855
nodeType, свойство, 316, 856
nodeValue, свойство, 856
Notation, объект, 870
notationName, свойство, 802
notations, свойство, 781
null, ключевое слово, 59
разрыв циклических ссылок, 205
проверка существования объектов, 229
Number, объект, 196, 543, 550
локализованные значения, 163
методы преобразования числа в строку, 196
Number(), вызов в качестве функции, 195

O

Object, объект, 157, 550, 557
hasOwnProperty(), метод, 164
isPrototypeOf(), метод, 165
propertyIsEnumerable(), метод, 164
toLocaleString(), метод, 163
toString(), метод, 161
получение значения класса, 162
valueOf(), метод, 163
наследование, 160
объекты, определяемые пользователем, 162
<object>, тег, 282, 283
Object, тип данных, 45
Object(), вызов в качестве функции, 195
offset(), метод, 667
offsetHeight, свойство, 628
offsetLeft, свойство, 628
offsetParent, свойство, 628
offsetTop, свойство, 629
offsetWidth, свойство, 629
offsetX, offsetY, свойства, Event, объект (Internet Explorer), 426, 614
onabort, обработчик события, 274, 641
onblur, обработчик события
Input, объект, 647
потеря окном клавиатурного фокуса, 713
элементы форм, 295

onchange, обработчик события, 29, 223
Input, объект, 648
Select, объект, 692
Text, объект, 697
Textarea, объект, 699
элементы форм, 295
FileUpload, 298
Radio и Checkbox, 297
Select, 299
поля ввода текста, 297
onclick, атрибут (обработчика события), 24
onclick, обработчик события, 29, 223, 280
Button, объект, 596
Checkbox, объект, 598
HTMLElement, объект, 632
Input, объект, 649
Label, объект, 263
Link, объект, 670
Reset, объект, 689
Submit, объект, 695
элементы форм, 295
Radio и Checkbox, 297
ondblclick, обработчик события, 632
onerror, обработчик события, 239, 274
Image, объект, 641
Window, объект, 713
подавление ошибок, связанных с версией языка, 443
onfocus, обработчик события, 715
Input, объект, 649
Window, объект, 715
элементы форм, 295
onhelp, обработчик события, 633
onkeydown, обработчик события, 298, 633
onkeypress, обработчик события, 298, 634
onkeyup, обработчик события, 298, 633
onload, обработчик события, 229, 273
Image, объект, 641
Window, объект, 715
onmousedown, обработчик события, 223, 635
Netscape 4, 433
onmouseout, обработчик события, 223, 280, 440
HTMLElement, объект, 636
Link, объект, 671
создание ролловера, 271
определение для Netscape 2 и 3 (Windows), 440
onmouseover, обработчик события, 223, 280
HTMLElement, свойство, 637
создание ролловера, 271
Link, объект, 672
onmouseup, обработчик события, 223, 635, 637
onmove, обработчик события, 716
onreset, обработчик события, 224, 287, 620
onresize, обработчик события, 716
onsubmit, обработчик события, 224, 287, 620
элемент <form>, 263
onunload, обработчик события, 229
Window, объект, 717
open(), метод
Document, объект, 260, 407, 608
документы, не содержащие HTML, 268
создание новых документов, 264
HTMLDocument, объект, 817
Window, объект, 234, 243, 407, 717
динамическое создание диалоговых окон с помощью методов объекта Document, 266
opener, свойство (Window), 233, 244, 700
решение проблем совместимости броузеров, 436
<optgroup>, тег, 300
<option>, тег, 681
Option, объект, 289, 299, 680
Option(), конструктор, 300
options[], массив, 299, 691, 693, 830
outerHeight, свойство, 701
outerHTML, свойство, 347, 629
outerText, свойство, 347, 629
outerWidth, свойство, 701
overflow, атрибут, 372
ownerDocument, свойство, 856
ownerElement, свойство, 738
ownerNode, свойство, 885
ownerRule, свойство, 762

P

Package, свойство (Window), 463
Packages, объект (Netscape), 461

packages, свойство, 701
pageX, свойство
Event, объект (Netscape), 431, 612
Layer, объект, 662
pageXOffset, свойство, 419, 431, 701
pageY, свойство
Event, объект (Netscape), 431, 612
Layer, объект, 662
pageYOffset, свойство, 419, 431, 701
parent, свойство, Window, объект, 233, 253
parentElement, свойство, 629
parentWindow, свойство, 602
parse(), метод, 503
parseFloat(), функция, 197, 557
parseInt(), функция, 197, 558
Password, объект, 25, 289, 298, 682
path, атрибут cookie, 306, 308
pathname, свойство
Link, объект, 668
Location, объект, 247, 674
parent, свойство, 700
parentLayer, свойство, 662
parentNode(), метод, 892
parentNode, свойство, 333, 856
Node, объект, 316
parentRule, свойство, 755, 757
parentStyleSheet, свойство, 755, 885
Perl
синтаксис регулярных выражений, 61
сравнение с JavaScript, 16
средства регулярных выражений, не поддерживаемые в JavaScript, 186
pixelDepth, свойство, 690
platform, свойство, 241, 678
Plugin, объект, 683
enabledPlugin, свойство, 676
plugins[], массив, 601, 678
pop(), метод, 486
port, свойство
Link, объект, 668
Location, объект, 674
POSITIVE_INFINITY, константа, 546
position, атрибут, значения, 365
pow(), функция, 540
prevValue, свойство, 848
prefix, свойство, 856
preventDefault(), метод, 409, 417, 805
и синтезированные события, 425
previousNode(), метод
NodeIterator, объект, 349, 868
TreeWalker, объект, 892
previousSibling, свойство, 327, 856
Node, объект, 316
previousSibling(), метод, 893
print(), функция, 129
print(), метод, 720
Window, объект, 234
primitiveType, свойство, 395, 750
ProcessingInstruction, объект, 774, 870
prompt(), метод, 233, 720
propertyIsEnumerable(), метод, 551, 554, 164
protocol, свойство
Link, объект, 669
Location, объект, 247, 674
prototype, свойство, 147
Function, объект, 527
функции, 138
функции-конструкторы, 160
publicId, свойство, 781, 802, 870
push(), метод, 209, 487

R

Radio, объект, 25, 290, 293, 296, 685, 687
random(), функция, 540
Range API, DOM, 349, 352
манипуляция областями, 354
начальная и конечная позиции области, 353
Range, объект, 352, 871, 883
представление одного узла или поддерева документа, 353
создание, 774
RangeError, объект, 559
RangeException, объект, 883
readOnly, свойство, 826, 842
readyState, свойство, 602
reason, свойство (IE Event), 614
Rect, объект, 395, 884
red, свойство, 885
ReferenceError, объект, 560
referrer, свойство, 261, 270, 601, 815
refresh(), метод, 679, 682
RegExp, объект, 61, 176, 189, 561, 566
методы для поиска по шаблону, 190
свойства экземпляра, 191
relatedNode, свойство, 848

relatedTarget, свойство, 419, 847
relative, позиционирование элементов, 366
releaseEvents(), метод, 721
reload(), метод, 249, 675
remove(), метод, HTMLSelectElement, 832
removeAttribute(), метод, 318, 346, 638, 797
removeAttributeNode(), метод, 797
removeAttributeNS(), метод, 798
removeChild(), метод, 316, 331, 862
removeEventListener(), метод, 809
    смешивание исходной модели события и DOM Level 2, 423
removeMember(), метод, 659
removeNamedItem(), метод, 852
removeNamedItemNS(), метод, 852
removeProperty(), метод, 760
replace(), метод
    Location, объект, 249, 675
    String, объект, 187, 575
        и свойство lastIndex, 191
replaceChild(), метод, 316, 863
replaceData(), метод, 334, 742
reset(), метод, 287
    Form, объект, 621
    HTMLFormElement, объект, 319, 824
Reset, кнопка (формы), 287, 620, 621
Reset, объект, 25, 290, 296, 687
resizeBy(), resizeTo(), методы,
    Layer, объект, 667
    Windows, объект, 234, 245, 722
return, инструкция, 41, 107, 119
returnValue, свойство, 427, 614
reverse(), метод, 171, 487
RGBColor, объект, 395, 884
Rhino (интерпретатор JavaScript на Java), 21
right, атрибут, 366, 368
right, свойство, 884
root, свойство, 866, 889
round(), функция, 541
routeEvent(), метод, 723
rowIndex, свойство, 838
rows, свойство, 834, 840, 842

S

\S (непробельный символ Unicode), 179
\s, пробел, метасимвол, 179
Screen, объект, 28, 243, 689
screen, свойство, 243, 700
screenX, свойство
    Event, объект, 612, 614
    MouseEvent, объект, 419, 847
    Window, объект, 701
screenY, свойство
    Event, объект, 612, 614
    MouseEvent, объект, 419, 847
    Window, объект, 701
<script> и </script>, теги, 22, 218
    language, атрибут, 85, 441
    броузеры, не поддерживающие теги <script>, 445
    включение файлов, 222
scroll(), метод, 28, 246, 723
scrollBy(), метод, 234, 246, 724
scrollIntoView(), метод, 638
scrollLeft и scrollTop, свойства (Internet Explorer), 419
scrollTo(), метод, 234, 246, 724
search, свойство
    Link, объект, 669
    Location, объект, 247, 674
search(), метод, 187
    String, объект, 576
    и свойство lastIndex, 191
sectionRowIndex, свойство, 838
<select>, тег
    onchange, обработчик события, 223
    Select, элемент, создание, 691
secure, атрибут cookie, 307, 308
Select, объект, 25, 290, 298, 690, 693
select(), метод
    HTMLInputElement, объект, 829
    HTMLTextAreaElement, 843
    Input, объект, 650
selected, свойство, 299, 829
    Option, объект, 681
selectedIndex, свойство, 830
    Input, объект, 643
    Select, объект, 299, 691
selectNode(), метод, 878
selectNodeContents(), метод, 879
selectorText, свойство, 394, 748, 761
self, свойство, 214, 700
    Window, объект, 233
separator, свойство, 744
setDate(), метод, 504
setFloatValue(), метод, 753
setFullYear(), метод, 504

setHours(), метод, 505
setInterval(), метод, 234, 238
setMember(), метод, 458, 659
setMilliseconds(), метод, 505
setMinutes(), метод, 506
setMonth(), метод, 506
setNamedItem(), метод, 853
setNamedItemNS(), метод, 854
setProperty(), метод, 391, 760
setSeconds(), метод, 507
setSlot(), метод, 458, 659
setStart(), метод, 881
setStartAfter(), метод, 882
setStartBefore(), метод, 882
setStringValue(), метод, 754
setTime(), метод, 507
setTimeout(), метод, 27, 228, 234, 238, 725
  Window, объект, 382
  очистка строки состояния, 440
setUTCDate(), метод, 507
setUTCFullYear(), метод, 508
setUTCHours(), метод, 508
setUTCMilliseconds(), метод, 509
setUTCMinutes(), метод, 509
setUTCMonth(), метод, 510
setUTCSeconds(), метод, 510
setYear(), метод, 510
setAttribute(), метод, 318, 346, 638, 798
setAttributeNS(), метод, 801
setAttributeNode(), метод, 799
setAttributeNodeNS(), метод, 800
setEnd(), метод, 880
setEndBefore(), метод, 881
setEndAfter(), метод, 880
setInterval(), метод, 27, 382, 724
sheet, свойство, 844
shift(), метод, 174, 487
shiftKey, свойство, 847
  Event, объект, Internet Explorer, 426, 614
  MouseEvent, объект, 419
short, тип данных, 735
sin(), функция, 48, 541
size, свойство, 826, 830
slice(), метод
  Array, объект, 172, 488
  String, объект, 577
sort(), метод, 134, 171, 489
source, свойство (RegExp), 191, 565
sourceIndex, свойство, 629
specified, свойство, 738
SpiderMonkey, интерпретатор JavaScript, 21
splice(), метод, Array, объект, 173, 209, 490
split(), метод, 209, 309
  String, объект, 578
splitText(), метод, 887
sqrt(), функция, 541
SQRT2, константа, 542
square(), функция, 54
src, атрибут, 222, 227
src, свойство, 826
  Image, объект, 639
  Layer, объект, 662
srcElement, свойство, Event, объект (Internet Explorer), 426, 614
srcFilter, свойство (IE Event), 614
startContainer, свойство, 872
startOffset, свойство, 872
static, атрибут, позиционирование элементов, 365
status, свойство, 232, 237, 700
  Window, объект, 726
stop(), метод, 727
stopPropagation(), метод, 409, 418, 806
String, объект, 566, 583
  HTML-методы, 567
  методы, получение значений cookie, 309
  определение метода для всех объектов класса, 150
String(), вызов в качестве функции, 195
String.fromCharCode(), метод, 426
<style>, тег, Netscape 4, использование JavaScript для определения таблиц стилей CSS, 225
Style, объект, 693
style, свойство, 318, 746, 748, 761, 819
  HTMLElement, объект, 629
StyleSheet, объект, 885
styleSheet, свойство, 746
StyleSheets, объекты, 393
styleSheets, свойство, 766
StyleSheetList, объект, 886
Submit, кнопка, 286, 287, 620, 621, 695
Submit, объект, 25, 290, 296, 694, 696
submit(), метод
  Form, объект, 287, 621
  HTMLFormElement, объект, 319, 825

substr(), метод, 579
substring(), метод, 209, 309, 580
substringData(), метод, 742
sun, свойство, 702
surroundContents(), метод, 882
switch, инструкция, 106
  в JavaScript, отличия от других языков, 108
  break, инструкция, 107, 114
SyntaxError, объект, 583
System, класс, 461
systemLanguage, свойство, 678
systemId, свойство, 781, 802, 870

T

tabIndex, свойство, 826, 830, 842
tagName, свойство, 629, 789
tan(), функция, 542
target, атрибут, 254, 622
target, свойство, 804, 871
  Event, объект, 417, 612
    Netscape, 431
  Form, объект, 286, 618, 622
  Link, объект, 669, 673
tBodies, свойство, 834
Tcl, язык, 21
<td>, тег, 833
test(), метод, 565
Text, объект, 25, 290, 297, 696, 698, 887
  создание, 775
  регистрация обработчиков событий, 411
text, свойство, 812, 829
  Anchor, объект, 281, 593
  Link, объект, 669
  Option, объект, 299, 681
text/html, MIME-тип, 268
<textarea>, тег, 698, 842
  onchange, обработчик события, 223
Textarea, объект, 25, 290, 297, 698
tFoot, свойство, 834
tHead, свойство, 834
this, ключевое слово, 97, 143, 144
  использование методов экземпляра, 152
  методы класса и, 153
  обработчики событий и, 295, 405
    addEventListener(), метод, 412
    ссылки на объекты-слушатели, 413
  обращение к глобальному объекту, 530
throw, инструкция, 119
timeStamp, свойство, 804
  Event, объект, 417
title, свойство, 815, 819
  Document, объект, 261
  HTMLElement, объект, 318, 629
  StyleSheet, объект, 885
<title>, тег, 601
toDateString(), метод, 511
toElement, свойство, 426, 614
toExponential(), метод, 196, 546
toFixed(), метод, 547
ToggleButton, класс (пример), 275
toGMTString(), метод, 308, 511
toLowerCase(), метод, 87
toLocaleLowerCase(), метод, 581
toLocaleString(), метод
  Array, объект, 175, 491
  Date, объект, 512
  Number, объект, 548
  Object, объект, 163, 551, 555
toLocaleUpperCase(), метод, 581
toLowerCase(), метод, 582
top, атрибут, 366, 368, 884
  Layer, объект, 387, 662
  Window, объект, 233, 253, 700
toPrecision(), метод, 196, 548
toString(), метод, 48, 81, 472
  Array, объект, 175, 491
    Netscape 4, реализация JavaScript 1.2, 209
  Boolean, объект, 493
  Date, объект, 513
  Error, объект, 518, 520
  Function, объект, 528
  JSObject, объект, 660
  Number, объект, 549
  Object, объект, 161, 551, 555
    Netscape 4, реализация JavaScript 1.2, 209
  Range, объект, 883
  RegExp, объект, 566
  String, объект, 582
  преобразование
    объектов в числа, 194
    чисел в строки, 196
toTimeString(), метод, 513
toUpperCase(), метод, 582
toUTCString(), метод, 513

toUpperCase(), метод, 87
Traversal API, DOM, 349
NodeIterator, объект, 349
TreeWalker, объект, 350
фильтрация, 350
TreeWalker, объект, 349, 888
создание, 775
фильрация, 350
true и false, логические значения, 45, 53
преобразование в другие значения, 54
try/catch, инструкции, 121, 518
finally, инструкция, 120, 122
language, атрибут, установка, 442
type, атрибут, 219
type, свойство, 288, 755, 804, 826, 830, 842, 886
CSSRule, объекты, 394
Event, объект, 417, 613, 614
Internet Explorer, 426
Netscape, 431
Input, объект, 643, 650
MimeType, объект, 677
Select, объект, 691
типы элементов формы, 651
элементы форм, 294
TypeError, объект, 494, 583
typeof, оператор, 60, 63, 95
применение к строковым значениям, 166
проверка типа данных аргумента, 129

U

UIEvent, объект, 418, 893
интерфейсы DOM, события, UIEvent, 414
undefined, значения, 59
генерация с помощью оператора void, 98
идентичность, 84
неприсвоенные и, 69
свойства, 142
undefined, свойство, 528
unescape(), функция, 307, 585
Uniform Resource Locators (см. URL), 609
Unicode, 39
keypress, события, 426
в идентификторах, 43
в строковых литералах, 50
управляющие последовательности, 51
unshift(), метод, 174, 492
uppercase(), функция, 332
URI
decodeURI(), функция, 515
decodeURIComponent(), функция, 515
encodeURIComponent(), функция, 517
encodeURI() , функция, 516
URIError, объект, 585
URL
about, ограничение по безопасности, 451
javascript, спецификатор псевдо-протокола, исполнение кода, 218, 230
Location, объект и, 674
ограничения по безопасности на подтверждение форм, 450
текущий отображаемый документ, 248
URL, свойство, 609, 815
use, ключевое слово, 44
useMap, свойство, 826
User-Agent, поле (HTTP), 440
userAgent, свойство, 241, 678
userLanguage, свойство, 678
UTC(), метод, Date, объект, 514

V

value, свойство, 738, 827, 829, 842
Button, объект, 594
Button, элемент формы, 295
Checkbox, объект, 297, 597
FileUpload, объект, 615
ограничения по безопасности, 450
Hidden, объект, 300, 624
Input, объект, 643, 651
Option, объект, 299, 681
Radio, объект, 297, 685
Reset, объект, 687
Submit, объект, 694
Text, объект, 696
Textarea, объект, 698
valueOf(), метод, 81, 471
Boolean, объект, 493, 494
Complex, класс (пример), 163

Date, объект, 515
Number, объект, 550
Object, объект, 163, 551, 556
String, объект, 582
преобразование объектов в числа, 194
var, инструкция, 65, 116
VBScript (Visual Basic Scripting Edition), язык, 20, 219
verify(), функция, 301
view, свойство, 418, 893
visible, свойство, 701
visibility, свойство, 387, 662
vLink, свойство, 812
vlinkColor, свойство, 260, 601
Document, объект, 269
void, оператор, 60, 98, 224
void, тип данных, 735
vspace, свойство, 274, 639

W

\W (нетекст), символ ASCII, 179, 184
\w (текст), символ ASCII, 179, 184
W3C, поддержка броузером, 19
whatToShow, свойство, 866, 889
which, свойство, 431, 613
while, циклы, 109
continue, инструкция, 115
width, свойство
Document, объект (Netscape), 602
Event, объект, 613
Image, объект, 274, 639
Screen, объект, 243, 690
width(), метод, 368
Window, объект, 25, 72, 214, 232, 699
History, объект, 250, 450
Location, объект, 25, 247
open(), метод, 243
Navigator, объект, 240
Screen, объект, 28, 243, 689
screen, свойство, 700
время жизни, 230
время запуска и интервал, 238
методы создания диалоговых окон, 234
методы управления, 243
close(), 245
focus() и blur(), 245
moveTo(), moveBy(), 245
resizeBy(), 245
resizeTo(), 245
scrollBy(), scrollTo(), 246
перемещение окна (пример кода), 246
методы
navigate(), Internet Explorer, 704
Netscape, 703
ограничения по безопасости на закрытие, 450
перехват событий (Netscape 4), 432
работа с несколькими окнами и фреймами, 252, 253
JavaScript во взаимодействующих окнах, 255
цветные фреймы (пример), 257
свойства, 232
Internet Explorer, 702
Netscape, 701
пакеты, 463
строка состояния, 237
элементы, 719
window, свойство
Layer, объект, 662
Window, объект, 72, 214, 233, 701
Windows Scripting Host, использование интерпретатора JScript, 21
with, инструкция, 123
World Wide Web Consortium (W3C), 12
Document, объект, 22, 218, 260, 610
динамическое создание документа, 264, 279
HTMLDocument, объект, 817
стандарт DOM, 11, 19, 261, 314
Level 1 и 2, 731
writeln(), метод
Document, объект, 260, 611
HTMLDocument, объект, 818

X

X- и Y-координаты курсора мыши, 419
x, y (свойства)
Event, объект, 613, 614
Layer, объект, 662
XML
<!DOCTYPE>, 766
CDATASection, 739
DOM-представление документов, 315
DTD
DocumentType, объект, 781
объявление нотации, 870

сущности, 802
использование DOM, 342
пространства имен, 769
стандарт W3C DOM, 262

Z

z-index, атрибут CSS, 370
zIndex, свойство, 387, 663

А

абзацы, преобразование в верхний регистр, 333
адресная панель, видимость в окне, 701
алфавитный порядок, 89
сортировка массивов, 171
альтернативы в регулярных выражениях, 181
анализ HTML-кода, 226
анимация, 27
DHTML
изменение цвета, 382
кросс-платформенный сценарий, 389
перемещение кнопки Button по кругу, 383
замена изображений, 272
автоматический запуск в обработчике события onload, 273
реализация ToggleButton, 275
строка состояния, 238
цвет фрейма, 257
анонимные функции, 130
апостроф (’), в строках, заключенных в одинарные кавычки, 50
апплеты, 282
JavaScript, взаимодействие с, 27
JSObjects, использование в, 459, 656
использование в сценариях, 455
аргументы, 525
функции, 129
проверка количества, 135
арифметические операторы, 27, 48, 81
арккосинус, 535
арксинус, 535
арктангенс, 535
ассоциативность операторов, 80
ассоциативные массивы, 56, 158
индексация, 58
объекты, 158
хранение функций, 133
атрибуты
HTML
checked, 297
name, 293
элементы, 293
конфликты имен с ключевыми словами JavaScript, 319
value, 688
видимости, 371
overflow и clip, 372
модификация HTML-элементов с помощью стилей CSS, 25
обработчики событий, 263
свойства cookie, 310
стилей, 357
z-index (порядок элементов в стеке), 370
объединение, 361
объектные типы DOM, 395
соответствующие им свойства CSS, 375
стиль CSS, 357, 744
и объект Style, 694
объектные типы DOM, 395
удаление из элемента, 797
элементы DOM, 318

Б

базовые типы данных
объекты-обертки, 62
преобразование значений из одного типа в другой, 62
преобразование объектов в, 85
ссылочные типы и, 69, 199
числа, строки и логические, 45
безопасность, 448
cookie и, 307
JavaScript и, 28, 448
зоны безопасности и подписанные сценарии, 452
ограничения, 450
Password, элемент, значения, 298
взаимодействие Java-апплетов с JavaScript, 460
ограничения, 279
чтение/запись файлов, 28
бесконечность, деление на 0, 81
бинарные операторы, 79
блоки инструкций, 68, 102
область действия переменных на уровне блоков, 68

броузеры, 11, 213
cookie, 308
ограничения при сохранении, 308
JavaScript в URL, 224
Mozilla, 11
Navigator, объекты, 27, 240, 677, 680
возможности клиентского JavaScript для разных броузеров, 19
вопросы безопасности, 29
вопросы совместимости, 436
не-JavaScript броузеры, 445
решение, 436
действия по умолчанию, связанные с событиями, 409
исполнение программ JavaScript, 226
история просмотра, 449, 626
клиентский JavaScript, 17, 19
кросс-платформенный анимационный сценарий DHTML, 389
отображение домашней страницы, 711
перекрытие таблицы стилей по умолчанию, 392
поддержка DOM Traversal API, 349
поддержка разных версий CSS, 363
последние разработки, 11
совместимость со стандартом DOM, 320
среда программирования, 213
иерархия объектов и DOM, 215
окно как глобальный контекст исполнения, 214
событийная модель, 216
управление, 25
установка свойства bgColor, 269
четвертого поколения, DHTML, 385
буквы в именах идентификаторов, 43

В

ввод, проверка формы, 26
веб-броузеры, 19
веб-сайты,
примеры из этой книги, 13
Mozilla, интерпретатор JavaScript с открытым исходным кодом, 21
веб-серверы
ограничение безопасности, ослабление внутри домена, 261
ограничения при сохранении cookie, 308
веб-страницы
загрузка, 249
исполняемое содержимое (клиентский JavaScript), 22
версии
JavaScript, список версий с описаниями, 18
совместимость, 441
указание в атрибуте language, 220
явная проверка, 442
клиентского и базового JavaScript, 20
верхние колонтитулы HTML-таблиц, 836, 837
верхний регистр (см. регистр), 582
взаимодействие с пользователями, 26
взаимодействующие окна, 255
видео, проигрывание с веб-страницы, 284
видимость, cookie, 306
включение файлов JavaScript, 222
вложенные динамические элементы, 367
вложенные массивы, 170
вложенные объектные литералы, 57
вложенные функции, 129
анимация на основе CSS, 384
лексический контекст и, 206
обработчик событий, событийная модель Netscape 4, 433
объекты-слушатели и, 413
вложение, инструкция if с блоком else, 104
внеэкранные изображения и кэширование, 272
возврат каретки, 40
возвращаемые значения обработчиков событий, 404
восьмеричные числа, 47
управляющие последовательности для символов Latin-1, 51
время жизни
cookies, 306
переменная, 230
время запуска
очистка, 238
установка, 238

время и дата, 27
Date, объект, манипуляции, 494, 515
разбор строк даты и времени, 503
с локальным временем, 512
с миллисекундами, установка, 507
с универсальным временем (UTC), 495
время ожидания, 27
очистка, 707
установка, 725
«всплывание»
обработка событий, 409
при распространении события, событийная модель IE, 427
всплывающие диалоги, 24
вставка новых элементов в массив
unshift(), метод, 174
push(), метод, 173
вставка текста (в узел Text), 334
встраиваемые стили
вычисляемые стили и, 391
перекрывающие стили и, 393
встроенные изображения, 27
встроенные объекты, конструктор, 140
встроенные элементы, 72
входные данные, проверка формы, 301
вывод результатов с помощью метода alert(), 35
вызов методов, 145
вызов функций, 45, 55, 128
Array(), конструктор, 167
call() и apply(), 139
JavaScript из Java, 474
использование круглых скобок, 56
как методов объекта, 524
обработчики событий, 410
порядок, 411
планирование или отмена после установки задержки, 234
с помощью обработчиков событий, 228
выполнение, задержка сценария, 221
выражения
вычисление, 76
ассоциативность операторов и, 80
инструкциии и, 101
return, 119
операторы и, 77
приоритет операторов, 80
литералы и значения переменных, 76
операторы и типы данных операндов, 79
выход из циклов, 113
вычисления, 27
отображение чисел Фибоначчи, 35
платежи по ссуде (пример программы), 29
вычисляемые стили, 391, 738

Г

гистограммы, динамические (пример), 379
гиперссылки, 278
глобальная область действия, 134
функции и, 206
функции-обработчики событий и, 407
функции, созданные с помощью Function(), 208
глобальный объект, 72, 528, 531, 532
функции-конструкторы для классов, 529
глобальные методы, 153
глобальные переменные, 72
undefined, 60
список предопределенных, 44
счетчики ссылок и, 206
глобальные свойства, 44, 153
глобальный поиск по маске, 185, 191, 561
год, 503
локальное время, 499, 504, 510
поле года (Date), минус 1900, 503
универсальное время (UTC), 502, 508
граница слова (b), 185
графика, 28
группировка в регулярных выражениях, 181

Д

дата и время, 60
модификации, Document, объект, 27
последнего изменения документа, 27, 261, 601, 607
добавление в HTML-документ, 264
часовые пояса, 501

установка даты окончания действия cookie, 307
двоичные числа, 92
декремент (– –), оператор, 42, 83, 102
десятичные знаки, задание при преобразовании числа в строку, 196
деление (по модулю), оператор % , 81
десятичные целые, 46
диалоговые окна, 24, 234
динамическое создание, 266
методы объекта Window, 233, 705
отображение результатов, 35
подсказка, 721
подтверждение, 709
динамическое создание документа, 264
длина массивов (Netscape 4, реализация JavaScript 1.2), 209
день
месяца
время UTC, 501
локальное время, 499, 504
недели
время UTC, 501
локальное время, 499
добавление текста (в узел Text), 334
доверенные веб-серверы в домене, 261
доверенные сценарии, настройка безопасности, 452
документы
веб-адрес текущего отображаемого документа, 674
загрузка и отображение нового документа, 676
повторная загрузка текущего, 675
не содержащие HTML, 268
дочерние узлы
DocumentFragment, 338
изменение родительского узла, 334
древовидная структура DOM, 315
обход, 326
Internet Explorer 4, 345
перемещение узлов по, 334

Е

единицы измерения, указание для свойств стилей CSS, 377, 394

Ж

жирный текст в узле или его потомках, 334

З

заголовок окна броузера, требование по безопасности, 451
задержка выполнения кода, 27
закладки
URL javascript, 225
отображение персональной панели в окне, 701
секретность списков, 449
замена текста в узле Text, 334
замыкание, 207
запись
свойств объектов, 141
элементов массива, 168
файлов, 28
зарезервированные слова, 43
защитное кодирование, 437
значения
null, 59
true и false, возвращение оператором логическое ИЛИ (||), 91
красного, зеленого и синего цветов, 269
массивов, 58
сравнение, 83
типы данных, 45
функции как, 55
функциональные литералы в программе, 55

И

идентичность значений null, 84
идентификаторы, 43
получение элементов по идентификатору, 776
чувствительность к регистру, 40
иерархия
HTML-документа, 315
интерфейсы событий DOM, 416
класс JavaPackage, 463
классов, 156
объектов клиентского JavaScript, 215
и класс JSObject, 457
извлечение отдельных символов из строк, 53
изображения, переключение и анимационные эффекты, 27
изящный отказ, несовместимость броузеров и платформ, 441

имена, 43
  броузера, 241
  иерархия JavaPackage, 463
  объекты документа, 262
  окон броузера, 253
  окон и фреймов, 253
  свойств в виде строк, 158
  соглашение о регистре для классов и объектов, 152
  соглашения в HTML, 319
имитация событий, 451
инициализация
  массивов, 58
  объектов, 140
  переменных цикла, 111
  свойств объекта, 147
  свойств объектов событий, 424
инкремент (––), оператор, 42, 82, 102
инструкции, 101
  break, 114
  continue, 115
  do/while, 110
  else if, 105
  else, 104
  for, 110
  for/in, с ассоциативными массивами, 159
  function, 117, 127
  return, 119
  throw, 119
  try/catch, 121
  try/catch/finally, 120
  try/finally, 122
  var, 116
  while, 109
  в URL JavaScript, 224
  в обработчиках событий, 223
  выражения, 101
  метки, 113
  обработки, 870
  присваивания, 101
  пустые, 124
  составные, 102
    внезапное прекращение выполнения, 103
  точка с запятой в качестве разделителя, 35, 41, 101
индексы
  ассоциативные массивы, 57
  выбранных опций, 691
  массива, 58, 166
    разреженного, 169
  строка и массив, начиная с 0, 53
интерпретаторы JavaScript
  версии JavaScript, совместимые с ECMA-262, 11
  встроенные в веб-броузеры, 19
  на веб-серверах, 21
  с открытым исходным кодом, 11, 21
интервалы, 238
  установка, 724
интерфейсы DOM, 316, 318
  event, MutationEvent, 419
  NodeList, 331
  Range, 352
  независимые от языка, 324
  событий, 413
    Event, 417
    MouseEvent, 418
    UIEvent, 418
исключения
  DOMException, 782
  EventException, 806
  генерация, 119
исполнение кода JavaScript, 226
  URL JavaScript, 230
  время жизни окон и переменных, 230
  обработчики событий, 227
  отложенное, 726
  периодическое, 724
  сценарии, 226
  функции, 227
исполняемое содержимое веб-страниц, 22
исходная событийная модель, 397
  смешивание со стандартной (DOM Level 2), 422
  события и типы событий, 398
итераторы(NodeIterator), 866, 868
итерации, 109

К

кавычки в строках, 50
капитализация, 89
каскадные таблицы стилей, 356
квадратный корень, вычисление, 48
клавиатурные события
  Event, объект события, 633
  свойства HTMLElement, 633
клавиши-модификаторы клавиатуры (события), 612

классы
HTML-элементы, 630
Error, 61
Java, 623
JavaClass, объект, 464, 653
JavaObject, объект, 465
JSObject, класс, 656, 660
непосредственное использование из JavaScript, 461
создание экземпляров с помощью оператора new, 465
иерархия, 156
комплексных чисел, определение, 155
методы, 153
объектов, 151, 162
определение конструкторов, 143
преобразование в строки, 161
прототипы и наследование, 146
прототипы и встроенные классы, 150
символов (в регулярных выражениях), 179
члены, 152
методы экземпляра, 152
свойства класса, 153
свойства экземпляра, 152
экземпляры, 152
элементов, применение правил стилей CSS, 362
клавиатурный фокус (см. focus (), метод), 645
клиентские модули проверки окружения, 439
клиентский JavaScript, 19, 22, 213, 255, 589
DOM, объединение сценариев с интепретатором JavaScript, 19
W3C DOM API и, 12
безопасность, 448
ограничения по, 450
взаимодействие с HTML-формами, 25
взаимодействие с пользователем, 26
возможности, 24, 25, 28
для разных броузеров, 19
встраивание в HTML-документы, 217
URL, 224
обработчики событий, 222
встроенный в броузеры, 17
глобальные переменные и функции, 43
информация о клиентском состоянии, 26
исполняемое содержимое веб- страниц, 22
написание сценариев, 34
ограничения, 28
среда броузера, 213
иерархия объектов и DOM, 215
окно как глобальный контекст исполнения, 214
событийная модель программирования, 216
управление внешним видом и содержимым документа, 24
чувствительность к регистру в объектах и свойствах, 40
ключевые слова
break, 114
case, 106
catch, 121
do, 110
event, 422
float, 377
function, 127, 256
switch, 108
this, 152
undefined, 60
var, 65
while, 110
конфликты с именами HTML-атрибутов, 319
невозможность использования в качестве идентификаторов, 43
список по ECMAScript v3, 43
чувствительность к регистру, 40
кнопки
Button, элемент формы, 295
Reset, 688
Submit, 695
анимация DHTML, 383
переключатели, 296, 685, 687
преобразование абзацев в верхний регистр, 333
код
защитное кодирование для решения проблем совместимости платформ, 437
кэширование, 222
отступы, 104

кодировка символов
URI, компоненты, 517
создание строки по кодам, 570
шестнадцатеричные управляющие последовательности, 520
коллекция, HTML-элементы, 813
колонтитулы HTML-таблиц, 836
комментарии, 42, 445
в коде JavaScript, 29
смешение с HTML, 446
скрытие сценариев, 445
создание узла Comment, 770
компиляция апплетов с JSObject, 460
константы, 49, 533
e, 537
Event, объект (битовая маска), 612
ln10, 538
ln2, 538
log10e, 539
log2e, 539
NaN, нечисло, 545
π, 540
SQRT1_2, 542
SQRT2, 542
значения свойства
code (DOMException), 782
eventPhase, 803
primitiveType, 395, 749
константные выражения, 76
определение с помощью свойств прототипа, 148
определяемые DOM-интерфейсами, 733
числовые, 543
MAX_VALUE, 544
MIN_VALUE, 545
NEGATIVE_INFINITY, 545
конкатенация строк, 52, 81
оператор +, 81, 89
преобразование элементов массива, 170
контексты исполнения, 73, 255
копирование
по значению, 199, 200
по значению и по ссылке, 203
по ссылке, 199, 200
строки, 202
координаты
курсор мыши, 419, 426
пикселы (при возникновении событий), 431
курсор мыши, X- и Y, 419, 426
кэширование
и внеэкранные изображения, 272
кода JavaScript, 222

Л

лексемы, 40
лексическая структура JavaScript, 39
идентификаторы, 43
ключевые слова, 43
комментарии, 42
литералы, 42
набор символов Unicode, 39
лексический контекст, и вложенные функции, 206
литералы, 42
в выражениях, 76
восьмеричные, 47
массивов, 42, 59, 167
объектные, 42, 57, 141
регулярные выражения, 61, 177
с плавающей точкой, 47
сравнение, 83
строковые, 50
функциональные, 55, 131
и конструктор Function(), 208
числовые, 46
шестнадцатеричные, 47
логические операторы, 90
логический тип данных, 45, 53
Boolean, объект, 62
преобразование значений true и false, 54
строковое представление объектов, 163, 175
сравнение со значением, 83
локализация
дата и время, 512
массивы, строковое представление, 491
объекты, строковое представление, 551, 555
строки, преобразование в
верхний регистр, 581
нижний регистр, 581
числа, форматирование для регионов, 548
локальная область действия, 134
локальное время, 494, 495, 498
GMT или UTC, отличие от, 501
год, 499

установка, 504, 510
дата, строковое представление, 511
день месяца или недели, 499
часы, установка, 505
локальные переменные, 66
сборка мусора и циклические объекты, 206
лямбда-функции, 56, 131

М

массивы, 45, 58, 166
DOM-объекты и, 324
Java, преобразование в JavaScript, 470
JavaArray, класс, 466, 652
ассоциативные, 56, 158
доступ к элементам, 99
индексация (регулярная и ассоциативная), 58
как ссылочные типы, 199
массивов, 170
объекты и, 166
операторы для, 99
поиск в строке с помощью регулярных выражений, 188
преобразование в числа, 194
присванивание элементам, 133
создание, 58, 167
сравнение по ссылке, 84
элементы, 166
добавление нового, 168
задание длины, 169, 483
чтение и запись, 168
математические функции, 27, 48
арифметические операторы, 48
месяцы
время UTC, 502
установка для даты, 510
локальное время, 500
установка, 506
метасимволы в регулярных выражениях, 177
метки
case и default, инструкция switch, 113
в инструкциях for и break, 115
методы, 55, 23, 133, 144, 170
Array, объект, 170
concat(), 172
concat(), 484
DOM-интерфейсы, 733
Document, объект, 260
Function, объект, 137, 139
Java
вызов из JavaScript, 455
использование в JavaScript, 467
join(), 170, 485
pop(), 486
push(), 209, 487
reverse(), 171
reverse(), 487
shift(), 174, 487
slice(), 172
sort(), 489
splice(), 173, 209, 488, 490
String, объект, для использования регулярных выражений, 186
toLocaleString(), 163, 491
toString(), 175, 209, 491
unshift(), 174, 492
Window, объект, 243
класса, 153
объекта, 161
определение и вызов, 145
определение для встроенных классов, 150
фабрики для создания объектов (DOM), 326
функции и, 145
экземпляра, 152
миллисекунды
время UTC, 502
установка для даты, 509
локальное время, 500, 505
представление даты в указанном объекте Date, 500
установка для даты и времени, 507
минуты
время UTC, 502
установка для даты, 509
локальное время, 500, 506
многомерные массивы, 170
доступ к объектам JavaArray, 467
многострочный поиск, 185
модели обработки событий, 397
DOM Level 2, 408
Internet Explorer, 425
Netscape 4, 431
модули
DOM, стандарт
проверка поддержки, 787

события, 413
и интерфейсы, 416
подключаемые, 283
проверка окружения, 439
таблицы стилей и CSS, 393
тестирование поддержки, 321
мониторы, размеры, глубина цвета, 28

Н

наборы символов, 39
charset, свойство (Document), 602
defaultCharset, свойство (Document), 602
навигационная панель, использующая объекты History и Location, 251
надклассы, 156
нажатие на элементы ввода, 649
наибольшее число, представимое в JavaScript, 544
наименьшее число, представимое в JavaScript, 545
наименьший общий знаменатель, подход к решению проблем совместимости, 436
настройка стратегии безопасности, 452
наследование, 16, 146
DOM-интерфейсы, 732
классы, клиентский JavaScript, 590
на базе классов
надклассы и подклассы, 156
наследование на базе прототипов и, 151
от объекта Object, 160
свойства объекта и, 552
неалфавитно-цифровые символы в регулярных выражениях, 178
неграница слова (\B), метасимвол, 185
недоверенные сценарии
настройка безопасности, 452
программы JavaScript и классы Java, 462
неименованные функции в виде литералов в программах, 55
неинициализированные значения переменных, 69
неопределенные значения, 584
несколько сценариев в документе, 218
нетипизированные переменные, 64
преобразование типов в LiveConnect, 467
нечисло, 198
нижний регистр (см. регистр)

О

область действия
with, инструкция и, 123
глобальный объект и, 530
на уровне блоков, 68
обработчиков событий, 405, 410
объектов вызова, 72, 134
переменных, 66, 74
функции, созданные с помощью конструктора Function(), 208
обновление переменных цикла, 111
обработка исключений, инструкция try/catch/finally, 120
обработчики событий, 23, 26, 217, 222, 396
Button, объект, 595
Document, объект и, 263
запись в документ в другом фрейме/окне, 265
определение обработчиков событий, 263
handleEvent(), метод и, 711
HTMLElement, объект, 630, 632, 638
Image, объект, 641
Input, объект, 647
Link, объект, 280, 670, 672, 669
onchange (объект FileUpload), 616
onsubmit и onreset, 287
this, ключевое слово и, 405
URL javascript, в качестве, 225
Window, объект, 704, 713, 717
возвращаемые значения, 404
для ошибок, 239
изображения, 273
исполнение, 227
как HTML-атрибуты, 400
как свойства, 401
назначение функции нескольким элементам, 403
непосредственный вызов функции, 403
область действия, 405, 410
регистрация, 409
объектов как обработчиков, 412
смешивание исходной событийной модели и DOM Level 2, 422
событийная модель IE, 428
функции, 410

элементы формы, 29, 294
объект, глобальный, 529
объект вызова, 134
объектно-ориентированное программирование
методы, 23
наследование, 16
объектные литералы, 57, 141
объектный контекст, использование строк, 62
объекты, 24, 28, 45, 56, 140
Array, 45
constructor, свойство, 160
Date, объект, 27, 46
delete, оператор, 98
Document, объект, 24
DOM
методы-фабрики для создания, 326
поведение как массивов JavaScript, 324
Error, 46, 61
Form, объект, и объекты элементов формы, 25
Function, 45, 137
JavaArray, 466
JavaClass, 464
JavaObject, 465, 654
JavaPackage, 463
Navigator, объект, 27
RegExp, 61
Screen, объекты, 28
Window, объект, 25
встроенные в броузеры, взаимодействие с JavaScript, 27
иерархия, 215
как ассоциативные массивы, 56, 158
классы, 151
иерархия, 156
определение класса комплексных чисел, 155
клиентский JavaScript, чувствительность к регистру, 40
массивы и, 166
методы, 55, 144, 161
методы экземпляра, 152
наследование от прототипов, 146
объект JavaObject, преобразование JavaScript, 471
оператор создания, 96
операторы доступа, 99
порядок вызова функций- обработчиков, 411
преобразование в базовые типы данных, 85
прототипы и наследование, 146
регистрация как обработчиков событий, 412
свойства, 56
переменные как, 72
перечисление, 112, 142
удаление, 142
установка и чтение, 141
экземпляра, 152
содержимое HTML-документа, 24
создание, 57, 140
с помощью объектных литералов, 141
с помощью функции- конструктора и оператора new, 143
соответствующие HTML- элементам, 401
сравнение на идентичность, 84
сравнение по ссылке, 84
строки и, 53
строковое представление, 551
установка свойств и элементов массивов из Java, 458
объекты вызова, 72, 134
как ссылочные типы, 199
преобразование в базовые типы, 193
преобразование в строки, 551
преобразование строк в, 555
сравнение, передача параметров и сравнение по ссылке, 200
счетчик ссылок, 205
элементарное значение, 551, 556
объекты-обертки, 62, 470
объекты-прототипы, 138, 160, 527, 551, 553
объявление, 65
динамическое создание (пример использования DOM), 338
переменных, 65
var, инструкция, 116
счетчик цикла, 111
элементов массива, 168
ограничения безопасности, 450
зоны безопасности и подписанные сценарии, 452
стратегия общего происхождения, 451

окна
верхнего уровня
время жизни, 231
открытие новых, 234, 705
свойства parent и top, 253
отображение с помощью CSS, 373
управление из JavaScript, 25
округление чисел
ceil(), функция, 536
floor(), функция, 537
round(), функция, 541
операнды, 79
операторы, 77, 81
–– (декремент), 83
– (отрицание, унарный оператор), 82
! (логическое НЕ), 91
!= (неравенство), 86
!== (неидентичность), 86
% (деление по модулю), 81
& (поразрядное И), 92
&& (логическое И), 90
* (умножение), 81
, (запятая), 98, 102
? (условный), 95
^ (исключающее ИЛИ, XOR), 92
| (поразрядное ИЛИ), 92
|| (логическое ИЛИ), 91
~ (поразрядное НЕ), 92
+ (конкатенация строк), 81, 89
++ (инкремент), 82
+= (сложение с присваиванием), 94
< (меньше), 86
<< (сдвиг влево со знаком), 92
<= (меньше или равно), 86
=== (идентичность), 83
> (больше), 86, 89
>= (больше или равно), 86, 89
>> (сдвиг вправо со знаком), 93
delete, 97, 102
in, 88
instanceof, 88
new (создание объектов), 96
typeof, 95
void, 98
арифметические, 48, 81
ассоциативность, 80
вызов функции, 100
доступа к массивам и объектам, 99
инкремент и декремент, 102
логические, 90
операторы
операнды, 79
отношения, 86
поразрядные, 91
правила определения
идентичности значений, 84
равенства, 84
преинкремента и постинкремента, 82
приоритет, 80
присваивание, 93
и l-значения, 79
проблемы с Netscape, 85
равенства и идентичности, 83
сдвиг, 92
сравнение значений null и undefined, 60
сравнения, 86
преобразование объектного типа данных, 194
типы данных операндов и возвращаемых значений, 79
строковые, 88
типы данных операндов, 79
условные, 95
операции поиска и замены, применение регулярных выражений, 61, 187
операционные системы, язык по умолчанию, 678
определение
методов, 145
обработчиков событий, 263
регулярных выражений, 176
альтернативы, группировка и ссылки, 181
задание позиции соответствия, 184
классы символов, 179
повторение, 180
символы литералов, 178
флаги, 185
свойств объекта, 142
функций, 54, 117, 127, 256, 524
конструктор Function() и функциональные литералы, 208
область действия, 207
с помощью конструктора Function(), 56
с помощью функциональных литералов, 55
собственные свойства, 138

опции
HTMLOptionElement, объект, 829
Select, объект, 690, 693
освобождение памяти, 71
открытые поля (апплетов), доступ из JavaScript, 455
отладка, for/in, цикл, 36
отложенное исполнение кода, 726
отрицательное условие на последующие символы, 185
отступы в коде, 40, 104
ошибки, 26
EvalError, объект, 522
onerror, обработчик события, 274, 641, 713
RangeError, объект, 559
ReferenceError, объект, 560
SyntaxError, объект, 583
TypeError, объект, 583
URIError, объект, 585
изящный отказ, 441
подавление ошибок связанных с версией языка, 443
проверка, 26
сооветствующие классы, 61

П

пакеты, Java, 462, 655
память, освобождение (сборка мусора), 71, 203
панель меню, видимость в окне, 701
перевод строки, 40
break, инструкция и, 114
в строках, 51
в строковых литералах, 50
перед лексемами, 40
передача параметров, 199, 200
по значению, 198, 200
по значению и по ссылке, 203
по ссылке, 198
разные значения, 201
строки, 202
переключатели, 296
и элемент Select, 299
переключение изображений, 27
перекрывающие таблицы стилей, 392, 778
переменные, 64
в цикле for/in, 112
время жизни, 230
глобальные, 66, 72
и счетчики ссылок, 206
значения null, 59
и несколько сценариев, 218
информация о броузерах, 27
как свойства объектов, 72
локальные, 134
неприсвоенные и неопределенные, 69
область действия, 66, 74
и инструкция with, 123
объявление, 65, 116
окружения, броузер, 27
сборка мусора, 204
счетчики, 109
типизированные/ нетипизированные, 64
преобразование данных в LiveConnect, 467
фреймы, различные, 255
цикла, инициализация, проверка, обновление, 111
чувствительность к регистру, 40
перемещение окна (пример кода), 246
перемещение по истории просмотра страниц, 25
перечисляемые свойства объекта, 113, 142, 164, 551, 554
перестановка узлов документа
в обратном порядке, 331
с помощью DocumentFragment, 337
с помощью рекурсии, 331
перетаскивание
модель событий DOM Level 2 (пример), 419
событийная модель IE (пример), 428
событийная модель Netscape 4 (пример), 433
перехват и распространение события, 408
addEventListener(), метод и, 410
перетаскивание элементов документа (DOM Level 2), 420
событийная модель Netscape 4, 432
персональная панель закладок, 701
пи, константа, 540
пикселы, единицы измерения, 368
платформы, вопросы совместимости, 436
обходные решения, 438
сбой, 441

серверные сценарии, 440
плюс или минус бесконечность, 81
повторение в регулярных выражениях, 180
\, обратная косая черта, 181
подклассы, 156
подключаемые модули, обновление массива plugins[], 679
подстроки, разбиение строк, 188
подтверждение форм, ограничения по безопасности, 450
подвыражения в регулярных выражениях, 182
позиционирование элементов с помощью CSS
z-index, атрибут, 370
атрибуты отображения и видимости, 371
видимость (частичная), атрибуты overflow и clip, 372
задание положения и размера, 366
пример позиционирования, 373
процесс стандартизации, 363
подписанные сценарии, 452
поиск веб-страниц, 279
поиск символа в строке, 52
поиск по маске (и регулярные выражения), 46, 61, 176, 190
RegExp, объект, 189, 561, 566
замена подстроки, 575
определение регулярных выражений, 176
альтернативы, группировка, 181
задание позиции соответствия, 184
классы символов, 179
повторение, 180
символы литералов, 178
флаги, 185
строки, поиск соответствий, 573, 576
строковые методы, 186
положительное условие на последующие символы, 185
получение элементов по именам тегов, 777
поля (апплетов), доступ из JavaScript, 455
поразрядные операторы, 91
порядок алфавитный, 89
порядок в стеке
слои, изменение, 665
элементы, 370
порядок выполнения операций, 80
последний символ строки, получение, 52
постоянство переменных, 66
представления (views), DOM, 737
предварительный просмотр событий до их обработки (Netscape), 432
предопределенные функции, 54
и глобальные переменные, 44
предопределенные элементы JavaScript
как свойства глобального объекта, 72
конструкторы, 140
преобразования
дат и времени, 60
объектов в строковые литералы, 161
переменных, 65
типов данных, 192
Java в JavaScript, 455
автоматические в JavaScript, 193
объектов JavaObject в элементарные типы JavaScript, 471
объектов в значения базовых типов, 193
работа с Java-типами из JavaScript, 467
строк в числа, 197
чисел в строки, 196
явные, 195
приоритет
оператор + (конкатенация), 89
операторов, 80
пробельные символы (см. символы-разделители)
проверка
аргументов функции, 135
версии JavaScript, 442
данных формы, 26, 301
наличия возможностей, 437
переменных цикла, 111
существования объектов, 229
программы JavaScript и секретность, 449
производительность, кэширование JavaScript-кода, 222

пространство имен
Attr, узел, 769, 800
Element, узел, 771, 778
атрибуты, 794
проверка, 796
удаление, 798
элемента, 795
прототипы, 146
isPrototypeOf(), метод, 165
встроенные классы, 150
наследование и, 146, 156
на базе классов, 151
объектов, 553
псевдослучайные числа, 540
пустые инструкции, 124
пустые массивы, 167

Р

разрыв циклических ссылок, 205
различие между значениями null и undefined, 60
размер
глубина цвета монитора и, 28
изображений, 274
окна, ограничения по безопасности, 451
экрана броузера, 243
разреженные массивы, 169
разрешающая способность изображения, 275
распространение событий, 408
DOM Level 2, 420
всплывание, 427
перехват, событийная модель Netscape 4, 432
регистр символов, 40
алфавитная сортировка и, 89
атрибуты обработчиков событий в HTML и XHTML, 401
верхний, преобразование строк, 581, 582
нечувствительность, в поиске по шаблону, 185
нижний, преобразование, 581, 582
чувствительность
HTML-теги, 329
в поиске по маске, 564
в регулярных выражениях, 191
к регистру в JavaScript, 40
свойства JavaScript, 401
соглашение об именовании в JavaScript, 152
регистрация обработчиков событий, 409
временно, 411
объекты в качестве обработчиков, 412
ограничения по безопасности, 451
смешивание исходной событийной модели и DOM Level 2, 422
событийная модель Internet Explorer, 428
событийная модель Netscape 4, 432
функции, 410
addEventListener() и ключевое слово this, 412
регулярные выражения, 61
замена соответствующих подстрок, 575
строки, поиск соответствия, 573, 576
строковые методы, 186
рекурсивная функция перестановки узлов, 331
родительские узлы, изменение для дочерних узлов, 334

С

сборка мусора, 71, 203
счетчик ссылок, 205
свойства
CSS2Properties, 744
HTML API (DOM), 319
глобальные, 44
класса, 153
клиентский JavaScript, чувствительность к регистру, 40
объекта, 56, 140, 151, 215
Anchor, 281
Cookie, 310
CSS2Properties, 376
delete, оператор, 98
Document, 260, 269, 599, 603
Event, 424, 612, 614
for/in, инструкция и, 112
Form, 286
Function, 137
HTMLElement, 628
Image, 274, 638
Input, 642

JavaScript-привязка DOM API, 324, 335
Layer, 660, 663
с эквивалентами в CSS, 387
Link, 279, 280, 668
Location, 673
Navigator, 241, 677
Screen, 689
Window, 232, 699, 702
апплеты, 282
аргументы, 134
вложенные данные, 284
глобальные, 528
иерархия JavaPackage, 463
имена как строки, 158
локально определенные, проверка, 551, 552
наследование от класса Object, 157
неопределенные, 142
перечисление, 142
обработчик события, 402, 408
перечисляемые, 554
присваивание функции, 55, 133
прототипы и наследование, 146
создание с помощью функции-конструктора, 57, 143, 160
соответствующие атрибутам стилей CSS, 375
удаление, 102, 142
установка и запросы, 141
установка из Java, 458
определяемые DOM-интерфейсами, 733
позиционирования, спецификация единиц измерения, 377
стилей в JavaScript, работа с объектом CSS2Properties, 377
функции, 137
prototype, 138
определение собственных, 138
экземпляра, 152, 191
экземпляра объектов RegExp, 191
элементов формы, 294
секунды
время UTC, получение, 503
установка, 510
локальное время, 500
установка в Date, 507
серверные сценарии, 440
сжатие массивов-аргументов функции
concat(), 172
splice(), вставка, 173
символ вставки, 179
символы
в диалоговых окнах, 234
декодирование, 515, 585
кодирование для URI, 516
в регулярных выражениях, 177
преобразование между JavaScript и Java, 469
символы-разделители, 40
\b, (символ «забой»), 180
в JavaScript, 40
метасимволы для регулярных выражений, 179
необязательные точки с запятой, 41
отступы в коде, 40, 104
перевод строки, 40, 51
табуляция, 40
синтезирование событий (DOM Level 2), 423
смежные узлы, 328
события, 217
captureEvents(), метод, 706
createEvent(), метод, 772
MutationEvent, объект, 848
releaseEvents(), метод, 721
routeEvent(), метод, 723
UIEvent, объект, 893
действия броузера по умолчанию в ответ на событие, 409
мыши, 280
Link, объект, 671
свойства HTMLElement, 635, 638
обработка, 396
типы, 398
модули DOM, 413
регистрация обработчика для, 410
событийное программирование, 216
совместимость
броузеры, не-JavaScript, 445
версии языка (JavaScript)
language, атрибут , 441
загрузка новой (совместимой) страницы, 444
подавление ошибок, связанных с версиями, 443
явная проверка номера версии, 442

платформ и броузеров, 436
защитное кодирование для решения проблем совместимости, 437
обходные решения проблем конкретных платформ, 438
проверка наличия возможностей, 437
сбой, 441
сортировка
алфавитная, 89
элементов массива в обратном порядке, 171
составные инструкции, 102
break, continue, return, throw, 103
состояние
клиентское, 26
локальных переменных и аргументов в области действия функции, 207
объект Window и его переменные, 230
сохранение cookie, 307
сообщения об ошибках, вывод с помощью метода alert(), 705
спам, 449
сравнение
возврат логических значений, 53
операторы равенства и идентичности, 83
по значению, 199, 200
строки, 203
по значению и по ссылке, 203
по ссылке, 199, 200
строковых литералов, 89
средства тестирования поддержки стандарта DOM, 321
ссылки
onmouseout, обработчик событий, 440
в массивах, 58
в регулярных выражениях, 181
и кнопки, 296
манипуляции со значениями, 198
передача по значению, 202
передача объекта функции-конструктору, 143
подтверждение пользователя, 403
сравнение на равенство, 84
счетчик для сборки мусора, 205
цвета
открывавшихся ссылок, 601
свойства объекта Document, 269, 599
циклические, 205
установка строки состояния, 237
ссылочные типы данных, 69
копирование, передача в качестве параметров и сравнение объектов, 200
стандартная модель событий, 408
буксировка элементов документа (пример), 419
интерфейсы и детализирующие свойства событий, 416
распространение событий, 408
регистрация обработчиков событий, функции, 410
синтезирование событий, 423
смешивание с исходной событийной моделью, 422
стандарты
ECMA-262 для JavaScript, 18
последние разработки, 11
предпочтение по сравнению с конкретными броузерами, 21
стек слоев, 660
стек, реализация с помощью массива, 486
стилевые атрибуты, именование в JavaScript, 376
столбцы, таблицы HTML, 834
страницы
отображение, 25
связь cookie со страницами, 306
строка состояния
окно броузера, 237
приемы анимации, 238
строки, 45, 49, 200
CSSPrimitiveValue, объект, 753
HTML-таблиц
вставка, 837, 841
удаление, 837, 841
String, объект, 62
ассоциирование значений со строками, 166
дата и время
GMT, 511
локальные, 511, 512
длина, 52, 572
значения HTML-атрибута обработчика события, 402
как индексы ассоциативных массивов, 57

строки
конкатенация, 52, 570
копирование и передача по ссылке, 202
методы
использования регулярных выражений, 186
класса String
split(), 209
substring(), 209
неизменность, 202
преобразование, 491
кодов символов в, 426
логические значения, 493
массивов в, 175
между JavaScript и Java, 470
объектов JavaObject в, 472
объектов в строки, 161, 163, 555
регулярных выражений, 566
чисел, 196, 197, 549
элементов массивов и конкатенация, 170
поиск начала/конца, 185
получение последнего символа, 52
сравнение, 87, 89
на идентичность, 84
по значению, 83, 203
строковые литералы, 50
управляющие последовательности в строковых литералах, 51
строки запроса, 248
строковые литералы, 513
преобразование чисел, 196
как имена свойств, 158
конкатенация, 89
операторы, 88
тип переменных, 70
цветовые свойства, 269
суффиксы имен файлов для MIME-типа, 677
suffixes, свойство, 677
существование объектов, 229
сценарии, 40
задержка исполнения, 221
исполнение, 226
написание, 34
несколько в документе, 218
скрытие от броузеров, 445

Т

таблицы HTML, 834, 838, 840
верхние колонтитулы, 836
нижние колонтитулы, 836
стилей и модули CSS в DOM, 393
столбцы, 834
шапка, 833
элементы ячеек, 833
табуляция, 40
теги
<embed>, 283
<fieldset> и <label>, 300
<input>, создание элементов Radio, 686
<object>, 283
текст
добавление, вставка, удаление или замена в узле Text, 334
жирный (в узле или его потомках), 334
цвет по умолчанию в документе, 600
тернарный оператор, 79, 95
тесты, наборы на совместимость со стандартами, 323
типы данных, 64
DOM, 734
Java, 464, 654
JavaClass, класс, 464
JavaScript, 16
Date, объекты, 498
Object, класс, 551
логический, 493
строки, 568
числа, 544
LiveConnect, 462
преобразование Java в JavaScript, 473
Object, Array и Function, 45
TypeError, исключительные ситуации, 583
базовые, 45
объекты-обертки, 62
и ссылочные типы, 199
логический, 53
массивы, 58, 166
JavaArray, класс, 466
объект, 140
значения null, 59
операндов, 79

типы данных
оператор typeof, 95
операторы и, 79
передача по значению и по ссылке, 198
преобразования, 192
Java в JavaScript, 455
автоматические, 193
объектов в значения базовых типов, 193
строк в числа, 197
чисел в строки, 196
явные, 195
с плавающей точкой, 27, 46
анализ при преобразовании строк в числа, 197
и JavaScript, 456
округление вниз до ближайшего целого, 537
строки, 49
функции и, 55, 132
типизированные языки, 65
типизированные/нетипизированные переменные, 64
преобразование данных в LiveConnect, 467
типы/типизация, 16
точность числа, 548
тригонометрические функции, 536

У

увеличение счетчика, управляющего циклом, 82
удаление
cookie, 308
Document, объект, 605
Option, объекты, 299
переменных, 66
свойств объекта, 142
текста (в узле Text), 334
элементов массива
pop(), метод, 173, 486
shift(), 174, 488
splice(), метод, 173, 490
узлы, 316
Attr, 739
NamedNodeMap, объект, 850
дочерние, 315
импорт, 779
обход, 326
объект Range, содержащий один узел, 353
перемещение по дереву документа, 334
перестановка, 331
родительские, 315
связанные с целевым узлом события, 419
смежные, 315
создание и добавление в документы, 335
фильтрация с помощью NodeIterator и TreeWalker, 350
укорачивание массива, 170
унарные операторы, 79
– (минус), 82
+ (плюс), 82
new, 96, 143
typeof, 95
void, 98
универсальное время (UTC), 494, 495, 498, 501
Date, объект, методы, 501, 503
год, установка, 508
даты, строковое представление, 513
месяц, установка, 510
минуты, установка для даты, 509
управление поведением броузера, 25
управляющие последовательности, 51
список, 51
уровни DOM, 216
условные операторы, 95, 103
установка времени ожидания, 228
установка начальной и конечной позиций области, 353

Ф

файлы
JavaScript, 222
чтение/запись, 28
фильмы на веб-странице, воспроизведение с помощью LiveVideo, 284
фильтрация
NodeIterator и TreeWalker, объекты, 350
NodeFilter, объект, 864
события изменения фильтра, 614
флаги в регулярных выражениях, 185, 191
форматирование текста в диалоговых окнах, 234

форма, 285
HTMLFormElement, объект, 823
передача данных, 694, 696
проверка входных данных, 301
функции, 401
содержащая все виды элементов, 290
элементы
checkbox, 598
FileUpload, 616
Hidden, 624
определение, 288
сценарии, 292
типы, 651
фокус
выведение из элемента ввода, 647
изменение, 697
обработчики событий onfocus и onblur, 295
передача фокуса элементу ввода, 650
получение окном, 715
потеря окна, 713
удаление из элемента, 645
фрагменты документа, 337, 770, 780
фреймы, 25, 252, 704
JavaScript во взаимодействующих окнах, 255
в клиентском JavaScript, 214
время жизни, 231
запись в документ в другом фрейме из обработчиков событий, 265
имена, 253
навигационная панель, использующая объекты History и Location, 251
отношения, 252
печать, 234
ссылки на окно Window верхнего уровня, содержащее, 233
цветная анимация (пример), 257
функции, 45, 54, 127, 144
Arguments, объект, 135
Function, объект
prototype, свойство, 138
методы, 139
return, инструкция, 119
switch, инструкция, 107
type of с расширенными возможностями, 162
анонимные, 130
вложенные, 129
лексический контекст, 206
вызов, 55, 102, 128
JavaScript из Java, 474
планирование или отмена после установки задержки, 234
глобальные, 528
имена, чувствительность к регистру, 40
исполнение, 227
как данные, 132
передача в качестве параметра другим функциям, 133
конструкторы, 529
Function(), 130
математические, 48, 533
методы и, 145
несколько сценариев и, 218
область действия, 134
обработчики событий, 217, 290
вызов, 410
непосредственный, 403
назначение одной функции нескольким элементам, 403
область действия, 405
регистрация, 410
событийная модель IE, 428
удаление, 411
объект Function, свойства, 137
объекты вызова, 72, 134
оператор (), вызов функции, 100
определение, 54, 127
конструкторы и функциональные литералы, 208
преобразование типов данных, 195
присваивание свойствам-обработчикам событий, 263
проверка форм, 401
различные фреймы, 256
сравнение на идентичность, 84
сравнение по ссылке, 84
типы данных и, 55
функции сравнения для сортировки массивов, 171
функции-конструкторы, 57, 140, 257, 529
Date(), 60
Function() и функциональные литералы, 208
prototype, свойство, 147, 160
вызов с помощью оператора new, 96

определение методов, 145
определение объектов-прототипов, 147
работа с несколькими фреймами/окнами и, 257
предопределенные и определенные пользователем в нескольких окнах, 257
функциональные литералы, 55, 131

Ц

цвета, 269
RGBColor, объект, 753, 884
для экрана броузера, 243
свойства объекта Document, 269
ссылок
активных, 599
неоткрывавшихся, 601
открывавшихся, 601
фон и передний план, 260
фреймы с цветной анимацией, 257
целевой узел события, 408
целые
анализ при преобразовании строк в числа, 197
в индексах массивов, 58, 168
в качестве операндов поразрядных операторов, 91
и JavaScript, 456
преобразование в строки, 48
цепочки областей видимости, 74
Window, как глобальный объект, 214
вложенные функции и, 413
фреймы и, 255
циклические ссылки, 205
циклы
continue, инструкция, 115
do/while, 110
for, 110
for/in, 36, 112, 159
перечисление свойств объекта, 142
while, 109
выход с помощью инструкции break, 114
с метками, 113
увеличение счетчика, 82
цифры, в идентификаторах, 43

Ч

часовые пояса, 501
часы
время UTC, 508
локальное время, 499, 505
числа, 45, 46
NaN, 532
Number, объект, 62
двоичные, 92
диапазон в JavaScript, 544, 545
значения Infinity, NaN и Number, 49
копирование, передача параметров и сравнение по значению, 200
округление, 536, 537, 541
операции, 48
определение класса комплексных, 155
преобразование, 203
в строки, 196, 549
между JavaScript и Java, 468
объектов JavaObject в, 471
псевдослучайные, 540
с плавающей точкой, 27, 46, 47
представление в JavaScript, 48
сравнение по значению, 83
шестнадцатеричные, 47, 197
в кодах символов Latin-1 или Unicode, 51
Фибоначчи, 35
члены класса, 152
методы класса, 153
методы экземпляра, 152
определение для класса Circle, 154
свойства класса, 153
свойства экземпляра, 152
чтение
cookie, 309
свойства объектов, 141
файлов, 28
элементов массива, 168
чувствительность к регистру, сравнение строк, 87

Ш

шапки таблиц HTML, 833, 836
удаление, 836

Э

экспоненты, pow(), функция, 540
экспоненциальная нотация, 47, 546
преобразование чисел в строку, 196
экземпляры, 152
методы, 152
свойства, 152
элементарные типы данных
Java
char, преобразование в число JavaScript, 470
JavaScript
преобразование объектов JavaObject, 471
преобразование чисел в Java-типы, 468
строки, 568
числа, 544
элементы, 215
DOM, атрибуты, 318
HTML
атрибуты обработчиков событий, 397
доступ из JavaScript, 25
назначение функции-обработчика события, 403
соответствующие объекты JavaScript, 401
id, атрибут, применение правил стилей к отдельным элементам, 362
Window, объект, 719
классов символов с отрицанием, 179
массива, 166
добавление новых, 168
задание длины, 169
методы, 170
неопределенные, 59
объекты JavaScript, установка из Java, 458
позиционирование элементов с помощью CSS, 365
присваивание функций, 133
чтение и запись, 168
поиск в документе, 328
IE 4 и DOM, 345
порядок в стеке, 370
формы, 26, 285
определение, 288
сценарии, 292

Я

язык
операционная система, язык по умолчанию, 678
предпочтительный для пользователя, 678
языки программирования
JavaScript, 17
типизированные и нетипизированные, 65
языки сценариев, 17
ActionScript, 21
language, атрибут, 219
встроенные в броузеры, 21
задание в виде MIME-типа, 219
якорные элементы, 593
регулярные выражения, 184
ячейки HTML-таблиц, 833
вставка, 839
удаление, 839

По договору между издательством «Символ-Плюс» и Интернет-магазином «Books.Ru – Книги России» единственный легальный способ получения данного файла с книгой ISBN 5-93286-055-3, название «JavaScript. Подробное руководство, 4-е издание» – покупка в Интернет-магазине «Books.Ru – Книги России». Если Вы получили данный файл каким-либо другим образом, Вы нарушили международное законодательство и законодательство Российской Федерации об охране авторского права. Вам необходимо удалить данный файл, а также сообщить издательству «Символ-Плюс» (piracy@symbol.ru), где именно Вы получили данный файл.